

①

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AD-A284 979



THESIS



**SOFTWARE FAULT TREE ANALYSIS
OF CONCURRENT ADA
PROCESSES**

by

William Samuel Reid, Jr.

September, 1994

Thesis Advisor

Timothy J. Shimeall

Approved for public release; distribution is unlimited.

94-31073



DTIC QUALITY CONTROLLED 3

94

9

28

1 10

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE Sep 1994		3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE SOFTWARE FAULT TREE ANALYSIS OF CONCURRENT ADA PROCESSES			5. FUNDING NUMBERS	
6. AUTHOR(S) Reid, William S.				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited			12b. DISTRIBUTION CODE A	
13. ABSTRACT (maximum 200 words) <p>The Automated Code Translation Tool (ACTT) was developed at Naval Postgraduate School to partially automate the translation of Ada programs into software fault trees. The tool works as follows: 1). The Ada parser and lexical analyzer calls the ACTT upon recognition of an Ada statement; 2) The ACTT produces a template representing the statement; 3). The templates are linked together.</p> <p>The tool was lacking in that it only looked at a subset of Ada structures. The problem that this thesis addresses is the implementation of the missing language structures, specifically, concurrency and exception handling, to allow the ACTT to handle all of the Ada structures.</p> <p>The result is a tool that takes the Ada source code and provides the analyst with a sequence of templates, and summary information to assist in incorporating hazard information for generating a fault tree.</p>				
14. SUBJECT TERMS Software Safety, Software Fault Tree Analysis, Software Testing			15. NUMBER OF PAGES 92	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

Approved for public release, distribution is unlimited

**Software Fault Tree Analysis
of Concurrent Ada Processes**

by

William S. Reid, Jr.
Lieutenant Commander, United States Navy
B.A., Saint Leo College, 1981

Submitted in partial fulfillment of the
requirements for the degree of

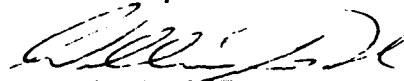
MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

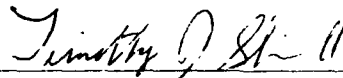
September 1994

Author:



William S. Reid, Jr.

Approved by:



Timothy J. Shimeall, Thesis Advisor



David Gaitros, Second Reader



Ted Lewis, Chairman
Department of Computer Science

ABSTRACT

The Automated Code Translation Tool (ACTT) was developed at Naval Postgraduate School to partially automate the translation Ada programs into software fault trees. The tool works as follows: 1). The Ada parser and lexical analyzer calls the ACTT upon recognition of an Ada statement; 2) The ACTT produces a template representing the statement; 3). The templates are linked together.

The tool was lacking in that it only looked at a subset of Ada structures. The problem that this thesis addresses is the implementation of the missing language structures, specifically, concurrency and exception handling, to allow the ACTT to handle all of the Ada structures.

The result is a tool that takes the Ada source code and provides the analyst with a sequence of templates, and summary information to assist in incorporating hazard information for generating a fault tree.

Accession For	
NTIS - GASI	<input checked="checked" type="checkbox"/>
DTIC - DB	<input type="checkbox"/>
Unclassified	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Special
A-1	

TABLE OF CONTENTS

I. INTRODUCTION	1
A. SOFTWARE SAFETY	1
B. SOFTWARE SAFETY ANALYSIS	2
1. Software Fault Tree Analysis	3
2. Petri Nets and Software Safety Analysis	6
C. PREVIOUS WORK	7
1. Failure Mode Templates	8
2. Automated Software Safety Analysis	8
3. Automated Code Translation Tool	9
D. TRANSPUTERS	10
E. STATEMENT OF PROBLEM	11
F. SUMMARY OF CHAPTERS	14
II. DEVELOPMENT	15
A. CREATION OF BASIC TEMPLATES	15
1. Entry Template	15
2. Select Template	15
a. Conditional Entry Template	17
b. Timed Entry Template	18
c. Selective Wait Template	18
3. Delay Template	19
4. Rendezvous Template	20
5. Abort Template	20
6. Additional Modifications	21
B. SHARED VARIABLES	21
C. REPRESENTATION CLAUSES	23
D. TRAFFIC LIGHT EXAMPLE	24
III. SOFTWARE ANALYSIS	33
A. SYSTEM BACKGROUND	33
B. SYSTEM DESCRIPTION	34
C. GENERAL SOFTWARE DESCRIPTION	35
D. SOFTWARE ANALYSIS	35

1. Localization	36
2. Isolation	37
IV. CONCLUSION	43
A. RESEARCH SUMMARY	43
B. RECOMMENDATIONS	44
C. FUTURE RESEARCH	45
APPENDIX A: GENERAL FIGURES	47
APPENDIX B: ADA STRUCTURE TEMPLATES	50
APPENDIX C: FTE FILE INFORMATION	77
APPENDIX D: PROJECT.A SOURCE LISTING	78
LIST OF REFERENCES	82
INITIAL DISTRIBUTION LIST	84

ACKNOWLEDGEMENTS

I would first like to thank Professor Shimeall for providing the stimulus for my selection of software safety as a research area, and for the foundation and focus he provided as my thesis advisor. He made the entire process both worthwhile and enjoyable, despite the long hours of programming. I would also like to thank LtCol David Gaitros for the insight he provided as my second reader, and for providing one of the most worthwhile learning experiences at Naval Postgraduate School.

Finally, I would like to thank my wife, Dierdre, for her unwavering support during our tour in Monterey

The Automated Code Translation Tool is available via anonymous ftp. Contact shimeall@cs.nps.navy.mil for information.

I. INTRODUCTION

The importance of a rigorous software testing program cannot be overstated. As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications. Experience has shown that errors are more common, or pervasive, and more troublesome, in software than in other technologies [Ref. 1]. In fact, products often fail in their first real use after being subjected to inputs and environments not anticipated by either the programmers or test planners.

A. SOFTWARE SAFETY

Safety has been defined as "freedom from those conditions that can cause death, injury, occupational illness, or damage to or loss of equipment or property" [Ref. 2]. Software safety then, can be considered as freedom from software-caused death, injury, damage to or loss of equipment or property. A safe system is one in which every state can be considered safe. This is an ideal condition, yet to be achieved. All methodologies presently in use in software development involve humans and humans are fallible.

A reliable system is one which, with a specified probability, will perform its intended function for a specified period of time under a set of specified environmental conditions. There is a clear distinction between reliability and safety. Reliability takes into account every possible software error, while safety is only concerned with those errors resulting in actual system hazards [Ref. 3].

With the number of software-based essential systems in transportation, industrial, consumer, and medical systems continually increasing, safety concerns are becoming a highly prevalent issue. Hardware reliability and quality have increased to the point that, the use of software is seen as the determining factor in any increase in the risk of error or failure. This is due to the complexity of software and the difficulty of validating it against any possible error [Ref. 4].

Software is inherently safe in isolation, since it alone can do no physical damage. It can only be considered correct or incorrect with respect to the system in which it is functioning. Program designers must communicate with system designers (and vice versa) to ensure that all or at least most of the situational hazards are identified, prepared for, and treated by the controlling software [Ref. 5]. For this reason, software and hardware must be treated as one entity for analysis purposes.

Unlike hardware, where we can specify tolerance values, we cannot speak of sensitivity to small errors in software. A single punctuation error can prove disastrous. Software safety is the tip of an iceberg called sound system engineering [Ref. 6].

B. SOFTWARE SAFETY ANALYSIS

The high "cost" of system errors that compromise life-critical functions provides the impetus for the development of tools, techniques, and methodologies that will aid in the identification and prevention of safety failures [Ref. 7]. Engineering methodologies for ensuring hardware safety have enjoyed greater success than efforts for techniques to assist

in the development of safe software systems. Some of the problems associated with safety verification include the difficulty of providing realistic test conditions and simulating hardware errors, transient faults, and system interfaces. Even when simulation is used, it is difficult to guarantee its accuracy.

Software safety analysis is very similar to hardware safety analysis. The analysis requires a representation of the program logic such as a detailed design and a list of safety failures to be analyzed. These failures can be derived from the safety requirements. The analyst should be presented with a flowchart, and the design of the system on which to perform the analysis. The goal of the analysis is to find the failure modes or conditions which are or could lead to the specified safety failures, or to show that the logic contained in the design is not likely to produce any safety failures [Ref. 8]. Techniques such as Software Fault Tree Analysis, Software Sneak Analysis, and Petri Net Analysis are listed in MIL-STD-882B [Ref. 2], as software safety analysis techniques.

1. Software Fault Tree Analysis

Computers have replaced mechanical devices in many safety critical systems. A logical step toward safety in software systems is to apply existing tools wherever possible. One such tool is fault tree analysis.

Fault tree analysis was developed at Bell Telephone Laboratories in 1962 by H.R. Watson. It was designed initially to be used for safety and reliability studies of a missile system. Engineers at Boeing further developed and refined the procedures, and

became the method's foremost proponents as a method of performing safety analysis of complex electromechanical systems [Ref. 9].

A fault tree is a graphic representation of the various parallel and sequential combinations of faults that result in the occurrence of the predefined event. These faults can be the result of component failures, human errors, or any other pertinent events and states that can lead to the hazard. A fault tree illustrates the logical interrelationships of basic events that lead to the undesired event which is the top event of the fault tree.

Software Fault Tree Analysis (SFTA) assumes that the system has failed in the way described by hazard analysis, and works backwards to determine the set of possible causes for the condition to occur. Taken another way, SFTA starts from the hazardous outputs (or lack of them) and traces backward to find paths through the code from particular inputs to these outputs or to demonstrate that such paths do not exist [Ref. 10]. At the root of the fault tree is the event which is to be analyzed, the loss event. Necessary preconditions are described at the next level with either an AND or an OR relationship. Each subnode is expanded similarly until all leaves describe events of calculable probability or are unable to be analyzed for some reason [Ref. 8].

Software fault trees are constructed using symbols from MIL-STD-882B. See Figure A-1. A *rectangle* is used to represent an event that requires further analysis. *Diamonds* are used for nonterminal events, which are not developed further for lack of information or insufficient consequences. A *circle* indicates an elementary event or primary failure of a component not requiring further development. The *house* is used to

represent normally occurring system events. *Ellipses* are used to indicate a state of the system that permits a fault sequence to occur. This may be a normal system state or a state resulting from system failure(s). A *triangle* represents another sub-tree for the node which is not depicted on the current tree. *AND* gates serve to indicate that all input events are required in order to cause the output event to occur, while *OR* gates indicate that any of the input events are satisfactory to produce the output event. These symbols comprise a subset of those used in hardware to facilitate integration between the hardware and software fault tree techniques.

Unlike hardware fault trees where hardware components fail independently of one another, software component failures are typically correlated. Even with the modern trend towards software modularity, it is doubtful whether the analysis of software trees will ever be as precise as for hardware trees. SFTA though, provides direct advantages to software analysis. These include:

- Provides the focus needed to give priority to catastrophic events and to determine the environmental conditions under which a correct state becomes unsafe.
- Provides a convenient structure to store the information gathered during the analysis which can be used later for redesign.
- The technique is familiar to hardware designers.
- Provides a single structure for specifying software, hardware, human actions, and interfaces with the system.
- Allows the examination of the effects of underlying machine failures or environmental changes versus verification techniques which assume system operates correctly.

Software Fault Tree Analysis provides extra assurance by focusing on hazards, by forcing a different view of the software, and by starting from different specifications [Ref. 8 & 10].

2. Petri Nets and Software Safety Analysis

One technique used in software safety analysis which has not been used to conduct hardware analyses is Petri nets [Ref. 11]. Designed primarily to be used for system modeling, Petri nets have been used to model and analyze systems for deadlock and reachability. They have been applied to hardware testing problems, protocol testing, network testing, and other areas, but their application to general software testing is still in its infancy [Ref. 12]. Using a systems approach, hardware, software, and human behavior can all be modeled using a single Petri net.

A Petri net structure is a 5-tuple consisting of a finite set of *places*, a finite set of *transitions*,¹ an input function mapping transitions to places, an output function mapping transitions to places, and the initial *marking* for the net. A Petri net graph is a directed multigraph representing a Petri net structure, whose nodes are transitions and places. It provides a convention for mathematical modeling of discrete event systems in terms of conditions and events and the relationship between them [Ref. 13]. Places model system conditions, and transitions model the occurrence of events. See Figure A-2.

Sequencing within a Petri net is controlled by the number and distribution of *tokens* in the net. Tokens reside in the places and control the execution of the transitions of the net. A Petri net executes by firing transitions. In firing, tokens are removed from input places and deposited in output places. The number of tokens contained in a place is

¹ The set of places and the set of transitions are disjoint.

called the marking of that place. The marking or state of the entire net consists of the set of markings of all the individual places within the Petri net. See Figure A-3.

Petri nets are an excellent analysis tool for describing and studying information processing systems characterized as being concurrent, asynchronous, distributed, parallel, non-deterministic, and/or stochastic [Ref. 14]. Individual processes can be represented by a Petri net. A composite net, the union of Petri nets for the individual processes, can represent the concurrent execution of the individual processes.

C. PREVIOUS WORK

The primary goal of formal techniques for software safety analysis is to ensure that a software system either satisfies a particular property or exceeds some property. Because humans are fallible, manual techniques can lead to the introduction of errors. Computerized aid is mandatory if any software is to be attacked using these formal techniques [Ref. 15].

McGraw [Ref. 16] investigated combining the methodologies of Petri Nets and Fault Tree Analysis for software safety analysis of an embedded military application. He proposed using the power of Petri nets to model concurrent systems in a multiple CPU environment. He concludes that Petri Net models can be applied to concurrent operations initially. Once the complex system is understood, Fault Tree Analysis can be used at points where the analyst feels that major problems are most likely to occur.

Gill [Ref. 14] presented a technique to convert and link Petri nets to fault trees and fault trees to Petri nets to take advantage of both analytical tools. Using Petri nets, the first step is to describe the system architecture. A shift could then be made to fault trees to describe the hazards associated with the system and the events that may lead to hazards.

1. Failure Mode Templates

Software Fault Tree Analysis is based on a series of templates that each map programming language constructs to a subtree. Cha, Leveson, and Shimeall [Ref. 10] discussed a manual method of performing SFTA for the Ada language. A software fault tree generated from these templates would provide a tree depicting where faults, if they occurred, could occur. The standard fault tree symbols discussed earlier are used in the construction of the templates. The Ada language templates are provided in Appendix B.

2. Automated Software Safety Analysis

Automated tools for software safety analysis provide analysts the opportunity to use their time more efficiently; focusing on the semantics of the analysis and not the syntax of the code. The existence of lexical analysis tools reduce the costs in developing automated tools to minimal.

Friedman [Ref. 9] described a tool to largely automate the process of constructing a software fault tree for a given Pascal program. The tool is designed to read in a Pascal program and a software caused hazard. It would then "fill in" template subtrees corresponding to the program's constructs. The output of the tool is an ASCII file

correctly formatted for TREE-MASTER, a commercial product, which can be used for displaying, editing, and printing the tree.

In his study of the application of Petri net modeling for safety analysis of a real-time military system (arming device for a guided missile system), Hayward [Ref. 13] determined that without the presence of automated tools, this type of analysis is impractical, even when applied to a small system. His paper suggests using automated tools during the creation of the Petri nets and to support queries for unsafe states.

A set of Petri Net Utilities (P-NUT) was developed at UC Irvine. Lewis [Ref. 17], using these utilities and the preexisting real-time system Petri net model developed by Hayward, examined the feasibility of applying automated software safety analysis to embedded military applications. The utilities proved awkward and difficult to use.

3. Automated Code Translation Tool

Using tools available at the Naval Postgraduate School (NPS), Ordonio [Ref. 18] developed an Automated Translation Tool that translates Ada code into a software fault tree. These tools included Aflex, an Ada based lexical analyzer, Ayacc, an Ada based parser generator, and Fault Tree Editor (FTE), an interactive fault tree design tool developed at NPS. Aflex and Ayacc were developed at the University of California, Irvine.

The tool consists of basically four components. The first component is a lexical analyzer. The lexical analyzer will determine if the given input consists of valid tokens. The next component is a parser. It will check the given input to ensure that valid Ada constructs are used. The third component is a template generator that transforms valid

statements into templates representing possible events associated with the statement in a format suitable for SFTA. The final component of the tool is a file generator that will create a file that meets the specifications of an FTE file type. See Appendix C for a description of FTE file format.

Fault Tree Editors are used to graphically display and modify fault trees. The Automated Translation Tool is limited to a subset of Ada structures; specifically, tasking and exception handling have not been addressed.

D. TRANSPUTERS

A transputer, derived from TRANSmitter and comPUTER, is a microcomputer with its own local memory and with links for connecting one transputer to another [Ref. 19]. Manufactured by Inmos Ltd., the transputer has two special features: an *on chip* serial link for communicating with other transputers, and hardware support for timesharing. The transputer can be used as a single chip processor or in networks to build high performance concurrent systems. An overview of Flynn's taxonomy for multiple CPU systems will reveal where transputer systems fit in [Ref. 20].

Flynn picked the number of instruction streams and the number of data streams as classification characteristics for CPU systems. This gives us four classification groups. Computers with a single instruction stream and single data stream are called SISD computers, those with multiple instruction, single data streams are referred to as MISC computers. All traditional single CPU computers are SISD, while presently, there are no

MISD computers. The next category is single instruction, multiple data stream computers (SIMD). Array processors fit into this category, where one instruction unit fetches an instruction and then commands many data units to carry it out in parallel, with different data sets. Finally, multiple instruction, multiple data stream computers (MIMD). All distributed computer systems are MIMD. SIMD and MIMD computers are further subdivided into two groups: shared memory systems and distributed memory systems. Systems built from transputers are MIMD and use distributed memory.

The transputer can be programmed in most high level languages, and is designed to ensure that compiled programs will be efficient. Transputers can also be used in single processor systems (SISD) as process controllers.

Several Navy tactical systems consist of computers with multiple processors. Consider for example the AEGIS combat system which uses the standard AN/UYK-7 computer with four processors. HP9000 series computers with two 68000 series processors are used onboard U.S. Navy submarines for Sonar Search Planning and Target Motion Analysis. In short time, such systems will not be able to handle the increasing demand for more complex software systems. Parallel processing systems are a must as the best high-performance uniprocessor architectures are reaching their limits [Ref. 21].

E. STATEMENT OF PROBLEM

The development of safe systems controlled by single CPUs is a complex and not well understood activity. The problem of ensuring the development of a safe system

becomes even more complex and error-prone in the case of distributed computing systems.

In a multitasking environment, a computer spends some unit of time (quantum) executing one task, switches its attention to other tasks at quantum expiration (or task completion), and eventually picks up where it left off on the original task if it has not been completed. This is referred to as interleaved concurrency. Overlapped concurrency occurs in multiprocessor environments, where different processors may execute different tasks at the same time.

Concurrent programming then is much more difficult than sequential programming. Not only must consideration be given to the relative speeds of different tasks, but deadlock is also a key factor. Debugging and testing a concurrent program can be agonizing because certain errors may depend on the timing of different tasks. The timing of tasks can vary from one execution of the program to another, so such errors might not be readily reproducible. Additionally, Ada programs have the property that sequences of statements are reentrant. In other words, several tasks may execute the same sequences of statements at the same time, and therefore different tasks executing the same statements manipulate copies of any variables within the scope of the sequence [Ref. 22].

Raising and handling exceptions in a *task* environment is similar to raising and handling them in subprograms with one important difference. If an exception were allowed to propagate out of a task, it would *asynchronously* interfere with its parent task, and in turn, it would be susceptible to the same kind of interference from tasks local to it. This

was therefore prohibited by the Department of Defense in its specifications for the Ada language. [Ref. 23]

The effect of raising an exception depends upon where the exception is raised; task declaration, task activation, during task execution, or during task communication. This turns out to be quite an important consideration. For example, exception **TASKING_ERROR** is raised only once, even if other exceptions are raised in the activation of many tasks. As previously stated, an exception raised in a task is not allowed to propagate out of a task. It can only be propagated out of a block or subprogram, but only after all dependent tasks have terminated.

It has been demonstrated repeatedly, that a balance between manual and automated techniques in SFTA should be applied during software analysis. The introduction of the Automated Code Translation Tool prototype was a step towards removing a large portion of the effort of code translation for Ada source code.

The initial prototype looked at only a subset of Ada structures, and as stated by Ordonio [Ref. 18], cannot be complete until all of the Ada structures are implemented. This thesis implements the Ada tasking structures absent from the original prototype, and introduces the exception handling templates. Several additional modifications were accomplished to further assist the analyst. These will be discussed in Chapter II.

F. SUMMARY OF CHAPTERS

Chapter II provides a summary of the development process, including modifications, and reasoning. It concludes with a comparison analysis of an Ada program used by Cha, Leveson, and Shimeall [Ref. 10]. Chapter III covers the modified tool and its application to the analysis of a military application. Chapter IV summarizes the research, indicating application and possible areas of future research. Appendix A contains background figures referenced in Chapter I. Appendix B contains the Ada structure templates defined by Cha, Leveson, and Shimeall with some modification.² Appendix C gives a description of the information required in a valid FTE file. Appendix D is the source listing for the application discussed in Chapter III.

² The first eighteen templates are reproduced from Reference 17.

II. DEVELOPMENT

A. CREATION OF BASIC TEMPLATES

To fully cover tasking in Ada, the following templates needed to be implemented:

- entry
- select
- selective wait
- select alternative
- conditional entry
- timed entry
- delay
- rendezvous
- abort, and
- exception.

1. Entry Template

Communication between tasks takes place via the actual parameters in the *entry call*, and the formal parameters in the corresponding *accept* statement during rendezvous. Entry statements without parameters are used for synchronization only. Entry families allow tasks to have multiple entries, that are all treated in the same manner. The *entry index* must be in the range specified in the entry family declaration. The conditions contributing to an entry call failure are then either a failure during the rendezvous, failure during parameter evaluation, or failure in the entry index.

2. Select Template

Three types of *select* statements exist: the *selective wait*, the *conditional entry call*, and the *timed entry call*. Conditional entry calls are used when an immediate rendezvous is desired. If the immediate rendezvous is possible, it takes place and the

sequence of statements following the entry are executed. If immediate rendezvous is not possible the alternative sequence of statements (*else* alternative) is executed. A timed entry call attempts to establish a rendezvous within a specified time. If the rendezvous is established within the specified period, the sequence of statements following the entry call are executed. If the rendezvous cannot be established with the specified period, the sequence of statements following the *delay* statement are executed.

The selective wait statement is a bit more complicated. The selective wait statement gives a task the capability to:

- accept entry calls from more than one task in a non-deterministic fashion,
- wait only a specified amount of time for an entry call to be made,
- perform an alternative action if no entry call is pending and
- indicate its readiness to terminate [Ref. 23].

It can have one or more alternatives followed by an optional else clause and sequence of statements. A *selective_wait_alternative* can be any one of the following:

- `accept_statement [sequence_of_statements]`³
- `delay_statement [sequence_of_statements]`, or
- **terminate**.

The presence of a terminate alternative precludes the presence of a delay statement.

Selective wait statements containing else clauses are prohibited from containing terminate or delay alternatives. Thus, a select failure can be attributed to a timed entry call failure, a conditional entry call failure, or a selective wait failure.

The select template has been modified substantially from that presented in Reference 10. The three possible alternatives contributing to a select failure have been separated into three distinct templates (selective wait, conditional entry, and timed entry)

³ Brackets ([.]) indicate optional occurrence of the item(s) contained within them.

to reduce tree expansion to only the applicable select grammar rule. An additional template (select alternative) was derived from the selective wait template. These changes also provide a closer relationship between the templates and the Ada grammar. See Figure 2-1

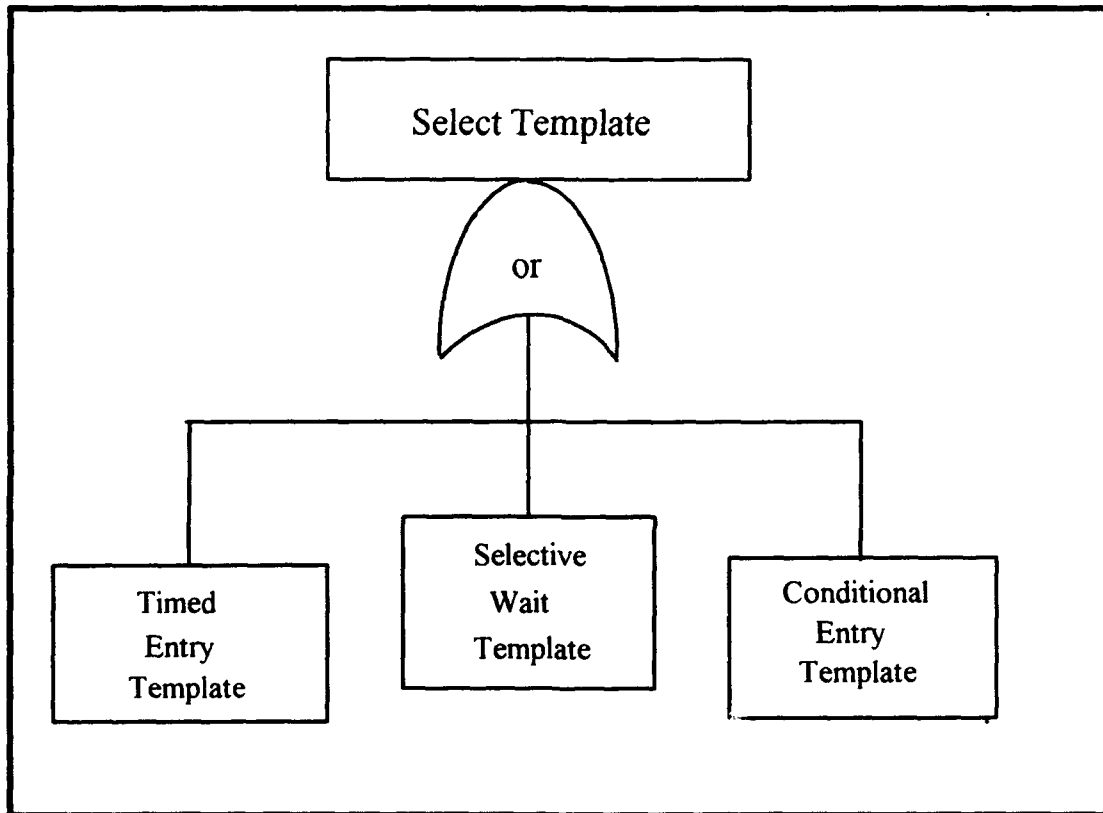


Figure 2-1 Select Template

a. Conditional Entry Template

As previously stated, conditional entry calls are used to attempt immediate rendezvous. Conditional entry calls are *cancelled* if they cannot be accepted immediately. To the called task, a cancellation has the same effect as if the call had never been issued.

The form of a conditional entry call is:

```
SELECT
    entry call statement
    [sequence of statements]
ELSE
    sequence of statements
END SELECT ;
```

Failure resulting from a conditional entry call may result from a failure in, or cancellation of the rendezvous, or a failure during the sequence of statements.

b. Timed Entry Template

If it is needed to establish a rendezvous within a specified time period, a timed entry call is used. A timed entry call is cancelled if it is not accepted within the specified time period. The form of a timed entry call is:

```
SELECT
    entry call statement
    [sequence of statements]
OR
    DELAY expression
    [sequence of statements]
END SELECT ;
```

If the entry call is not accepted within the time specified in the **DELAY** statement, it is cancelled and the sequence of statements following the **DELAY** statement are executed. Otherwise, the entry call, and the sequence of statements are executed. A timed entry failure may result from a failure during the rendezvous, the timeout, or the sequence of statements during the rendezvous.

c. Selective Wait Template

The selective wait statement has the following form:

```
SELECT
    ACCEPT statement
    [sequence of statements]
```

```

{ OR
    ACCEPT statement
    [sequence of statements/ ]4
[ELSE
    sequence of statements]
END SELECT ;

```

Each of the **ACCEPT** statements is referred to as a *select alternative* and is of the form:

```
[ when condition => ] selective_wait_alternative
```

Selective_wait_alternatives were previously discussed. A selective wait failure can then be caused by the select statement when the condition is evaluated as true, the select alternative, or the *else* sequence of statements.

3. Delay Template

Task execution can be temporarily suspended by use of a **DELAY** statement. The delay statement causes the executing task to remain inactive for *at least* the specified time. The task becomes *eligible* to resume execution upon expiration of the *duration*. The form of the delay statement is '**DELAY expression**' where expression belongs to a predefined fixed-point type named **DURATION**, representing time in seconds.

The capability may be desired for a task to perform some activity on a periodic basis, but also accept entry calls as they arise. A selective wait using a *delay alternative* will provide such a capability. A delay alternative is an alternative within a selective wait statement that begins with a delay statement instead of an accept statement. If not other alternative in the select wait is selected prior to expiration of the specified duration, then the delay alternative is executed.

⁴ Braces ({ .. }) indicate zero or more occurrences of item(s) contained within them.

Delay statement failures can thus be attributed to failures resulting from the task being delayed or evaluation of the type duration.

4. Rendezvous Template

Individual processes in Ada are referred to as tasks. The synchronization and subsequent communication between two tasks (one task *issuing* an *entry call* and the other task *accepting* the entry call) is referred to as a rendezvous.

Several tasks may desire rendezvous with the same task. If this condition exists, the rendezvous' will be queued and will occur in a FIFO order. So, at the outset, a rendezvous failure may occur during the rendezvous, or result from it not occurring. The conditions under which either of these may cause failure are straightforward and so the reader is referred to the figures for the Rendezvous Template in Appendix B.

Note the addition an *exception tree* to the **Task aborted** node. As depicted, the exception **TASKING_ERROR** will be raised if the called task completes before accepting an entry call or is completed at the time of the entry call, the called task becomes abnormal during rendezvous, or if an exception raised in the *accept statement* is not handled locally within an inner frame. [Ref. 23]

5. Abort Template

The **ABORT** statement is used to stop a task, preventing it from continuing communication or synchronizing with other tasks, thus rendering it *abnormal*. Any task dependent upon an abnormal task also becomes abnormal.

An aborted task does not necessarily terminate immediately. Only if a task is waiting at an entry call, an *accept* statement, a *select* statement, or a *delay* statement will

it terminate immediately. Otherwise, termination will occur when the task reaches a synchronization point; i.e., at the start or end of an accept statement, an exception handler. If a task is aborted in the midst of manipulation a data structure, data may be left in an inconsistent state. An aborted task does not have the opportunity to deallocate variables, close files, etc.

Aborting a task may result in failure if the program attempts to abort a task that has not yet been activated, or the task becomes abnormal while in a rendezvous. Abort failure may also result from task manipulation of variables during the abort.

6. Additional Modifications

Several record structures, modeled after those contained in the original source code, were added. These include a data structure for exception table records, and one for task entry table records. As was used in the original source for procedures and functions, the exception table record structure is used to keep track of all defined exceptions within the code, and the task entry structure is used to keep track of defined rendezvous'.

Since the subject of this thesis is centered upon the tasking facility of Ada, it was decided to generate a separate fault tree per task during parsing {separate FTE file generated} to afford the analyst the ability to examine each task body individually.

B. SHARED VARIABLES

Individual processes in Ada communicate through rendezvous. It is also possible for two tasks to communicate through a global (*shared*) variable whose scope includes both

tasks. This is hazardous because it may result in race conditions. Race conditions occur when concurrent tasks try to manipulate the same device, or update the same variable.

If two tasks access a shared variable, neither can assume anything about the order in which the other performs its operations, except at the start and end of their rendezvous. If the shared variables are scalar or access types the following rules must not be violated or unpredictable results may occur:

- If a task reads a shared variable between two synchronization points, then no other task must update this shared variable between these synchronization points; other tasks are allowed to read the shared variable.
- If a task updates a shared variable between two synchronization points, then no other task is allowed to read or update the shared variable between these two synchronization points.

Ada allows the programmer to designate the points at which a shared variable is either read or updated, as synchronization points through the use **pragma SHARED**. Pragas provide the compiler with information that may affect the way a program is listed, the way a program is translated into machine language, or the order in which a program performs certain actions [Ref. 22].

Since *tasking* commences at elaboration (immediately prior to *begin* of the main program) and is concurrent, those **IDENTIFIERS** associated with global procedures, functions, and variables, called or subject to manipulation by tasks are flagged for the analyst's consideration.

The original source code generated templates through *actions* placed in the Ayacc (ada.y) source file. To flag **IDENTIFIERS** during parsing of the source, actions had to

added to the Aflex (ada_lex.l) source file. This is due to the *consuming* nature of lexical analyzers; the actual text is not returned to the parser, but merely the token IDENTIFIER.

Storage of the IDENTIFIERS (those associated with procedures, functions, and variables) and keeping track of nesting levels for monitoring visibility required the addition of a stack package. Each IDENTIFIER is contained within a record structure which contains information as to whether the IDENTIFIER is a procedure, function, task, variable, or representation clause. The package prevents duplicate storage of identifiers and provides an output of those identifiers which may be *shared* after each task body is parsed. *Condition* identifiers, those found in if, when, and case statements, required the use of a separate stack package due to the nested nature of if-then-else/elsif statements.

C. REPRESENTATION CLAUSES

Communication with the underlying hardware is accomplished through the use of representation clauses. The types available are length clauses, enumeration clauses, record representation clauses, and address clauses.

Representation clauses are mechanisms provided by the Ada language by which a program is allowed to *control* internal representations. Such representations include the memory address or size of an entity, the amount of storage available for dynamic allocation or execution of a task, and the underlying representation of particular types. Representation clauses, along with representation attributes, allow a program to name implementation-dependant values abstractly, preserving portability [Ref. 22].

Since IDENTIFIERS are *popped* from the stack when a *scope* is closed, representation clauses are placed at the bottom of the stack to prevent them from being removed. Two stacks are used in the tool. A linked list was used to implement the stack for all identifiers except the *condition* identifiers. A generic stack package was used for storage of these.

D. TRAFFIC LIGHT EXAMPLE

During code development, the traffic light control system [Ref. 10] code was used extensively due to its tasking nature. The code appeared as follows:

```
1 procedure TRAFFIC is
2   type DIRECTION is (EAST, WEST, SOUTH, NORTH);
3   type COLOR is (RED, YELLOW, GREEN);
4   type LIGHT_TYPE is array(DIRECTION) of COLOR;
5   LIGHTS : LIGHT_TYPE := (GREEN, GREEN, RED, RED);
6   task type SENSOR_TASK is
7     entry INITIALIZE(MYDIR : in DIRECTION);
8     entry CAR_COMES;
9   end SENSOR_TASK;
10  SENSOR : array(DIRECTION) of SENSOR_TASK;
11  task CONTROLLER is
12    entry NOTIFY(DIR : in DIRECTION);
13  end CONTROLLER;
14  task body SENSOR_TASK is
15    DIR : DIRECTION;
16  begin
17    accept INITIALIZE(MYDIR : in DIRECTION) do
18      DIR := MYDIR;
19    end INITIALIZE;
20    loop
21      accept CAR_COMES;
22      if (LIGHTS(DIR) /= GREEN) then
23        CONTROLLER.NOTIFY(DIR);
24      end if;
```

```

25   end loop;
26   end SENSOR_TASK;

27   task body CONTROLLER is
28   begin
29     loop
30       accept NOTIFY(DIR : in DIRECTION) do
31         case DIR is
32           when EAST | WEST =>
33             LIGHTS := (GREEN, GREEN, RED, RED);    delay 5.0;
34             LIGHTS := (YELLOW, YELLOW, RED, RED); delay 1.0;
35             LIGHTS := (RED, RED, GREEN, GREEN);
36           when SOUTH | NORTH =>
37             LIGHTS := (RED, RED, GREEN, GREEN);    delay 5.0;
38             LIGHTS := (RED, RED, YELLOW, YELLOW); delay 1.0;
39             LIGHTS := (GREEN, GREEN, RED, RED);
40         end case;
41       end NOTIFY;
42     end loop;
43   end CONTROLLER;

44 begin
45   for DIR in EAST .. NORTH loop
46     SENSOR(DIR).INITIALIZE(DIR);
47   end loop;
48 end TRAFFIC;

```

We will now analyze this code, comparing the results produced by the tool, with the analysis presented by Cha, Leveson, and Shimeall.

The event analyzed was *two cars traveling north and east present in the intersection at the same time*. More specifically, a car traveling North is in the intersection (entered the intersection without stopping since the light was already green), as a car traveling East desires to enter the intersection. The fault trees developed by Cha, Leveson, and Shimeall for this condition are shown in Figures 2-6 and 2-7. The fault trees developed by the Fault Tree Generator are shown in Appendix B (if statement template and rendezvous template). Figure 2-8 depicts the tool generated fault tree for lines 33-35 which were determined by Cha, Leveson, and Shimeall to be the source of failure.

An analysis with the tool may have proceeded in the following manner. We first scan the output information produced by the tool. The first item we are informed of is that multiple instances of the task SENSOR_TASK exist. We can examine the code at this point and see that four instances of task type SENSOR_TASK are created (sensor.east, sensor.west, sensor.north, and sensor.south), one for each entrance to the intersection. See Figure 2-2.

```
Enter input file: ---- Starting Code Translation ----
1: procedure TRAFFIC is
2:   type DIRECTION is (EAST, WEST, SOUTH, NORTH);
3:   type COLOR is (RED, YELLOW, GREEN);
4:   type LIGHT_TYPE is array (DIRECTION) of COLOR;
5:   LIGHTS : LIGHT_TYPE := (GREEN, GREEN, RED, RED);
6:   task type SENSOR_TASK

** TRACK MULTIPLE INSTANCES OF TASK TYPE 'SENSOR_TASK' **
```

Figure 2-2 Multiple Task Instance Flag

After parsing the body of SENSOR_TASK, the tool provides a listing of IDENTIFIERS it has determined to be visible to the task. These are 1) TRAFFIC, SENSOR_TASK, and CONTROLLER; the main procedure and the two tasks, 2) LIGHTS, and SENSOR; global variables, and 3) MYDIR, and DIR; actual entry call parameters. An identical listing is output after the body of task body CONTROLLER is parsed. See Figure 2-3.

The summary information provided by the tool consists of a listing of the procedures, functions, and tasks, and the listing of possible interleavings (rendezvous'). See Figures 2-4 and 2-5.

```
The following IDENTIFIERS have been flagged
as global procedures, functions, and
variables that may be manipulated by this
task.
*****

TRAFFIC
LIGHTS
SENSOR_TASK
MYDIR
SENSOR
CONTROLLER
DIR
```

Figure 2-3 Global (Shared) Variable Output

```
The number of procedures, functions, and
tasks on the table are    3

The procedures, functions, tasks, and their
root faults on the table are the following:
-----

SENSOR_TASK
Sequence of statements caused fault

CONTROLLER
Sequence of statements caused fault

TRAFFIC
Sequence of statements caused fault
```

Figure 2-4 Subprogram Listing

We have only two subprograms to deal with, the tasks `SENSOR_TASK` and `CONTROLLER`, so we next examine the synchronization points. We see that *entry*

SENSOR_TASK.INITIALIZE is used only to assign a DIRECTION to each sensor, so it can be eliminated as a source of failure (source lines 45-47).

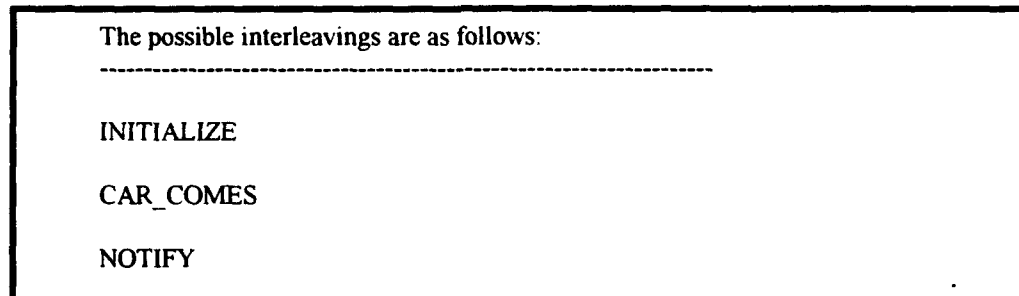


Figure 2-5 Rendezvous Points

Since the variable LIGHTS was flagged by the tool as being a shared variable, and due to the number of assignment statements, we first examine the body of task CONTROLLER. For the car traveling north to have entered the intersection without stopping, the condition of LIGHTS would have to be RED, RED, GREEN, GREEN. The code reveals that this condition exists for 5 seconds at the start of CONTROLLER.NOTIFY(SOUTH | NORTH), and at the completion of CONTROLLER.NOTIFY(EAST | WEST). LIGHTS will remain in this state until another vehicle enters the intersection. Since we know that the vehicle traveling north entered the intersection without stopping, no call was made to entry CONTROLLER.NOTIFY(SOUTH | NORTH), so we must either be at line 35 or have recently completed a CONTROLLER.NOTIFY(EAST | WEST) rendezvous.

Now looking at entry SENSOR_TASK.CAR_COMES (which makes the entry calls to CONTROLLER. NOTIFY) we note that there are no restrictions on when the call is *accepted* so the entry CAR_COMES will be processed immediately. The *if* condition is

examined, and a call will be made to entry CONTROLLER.NOTIFY if the *condition* is satisfied.⁵ More importantly, it is this *if statement* that allows the car traveling north to *bypass* the rendezvous [Ref. 10].

We can conclude then (as did Cha, Leveson, and Shimeall) that the hazard condition has the possibility of occurring if we have two successive SENSOR(EAST | WEST) rendezvous' with an intermediate north traveling car entering the intersection at the completion of the first SENSOR(EAST | WEST) rendezvous.

⁵ The null *else* implies no call to entry CONTROLLER.NOTIFY.

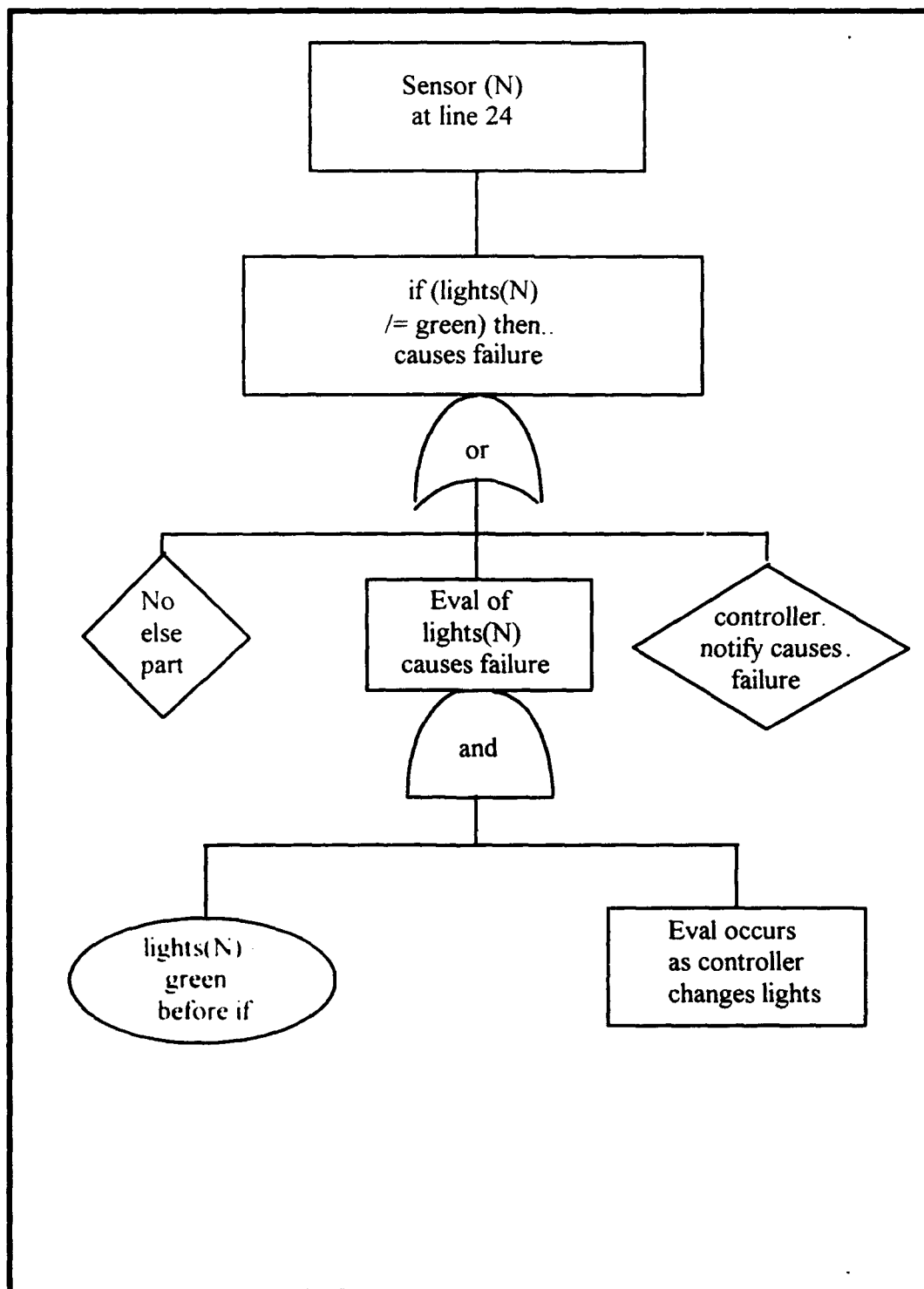


Figure 2-6 Sensor(North) at Line 24

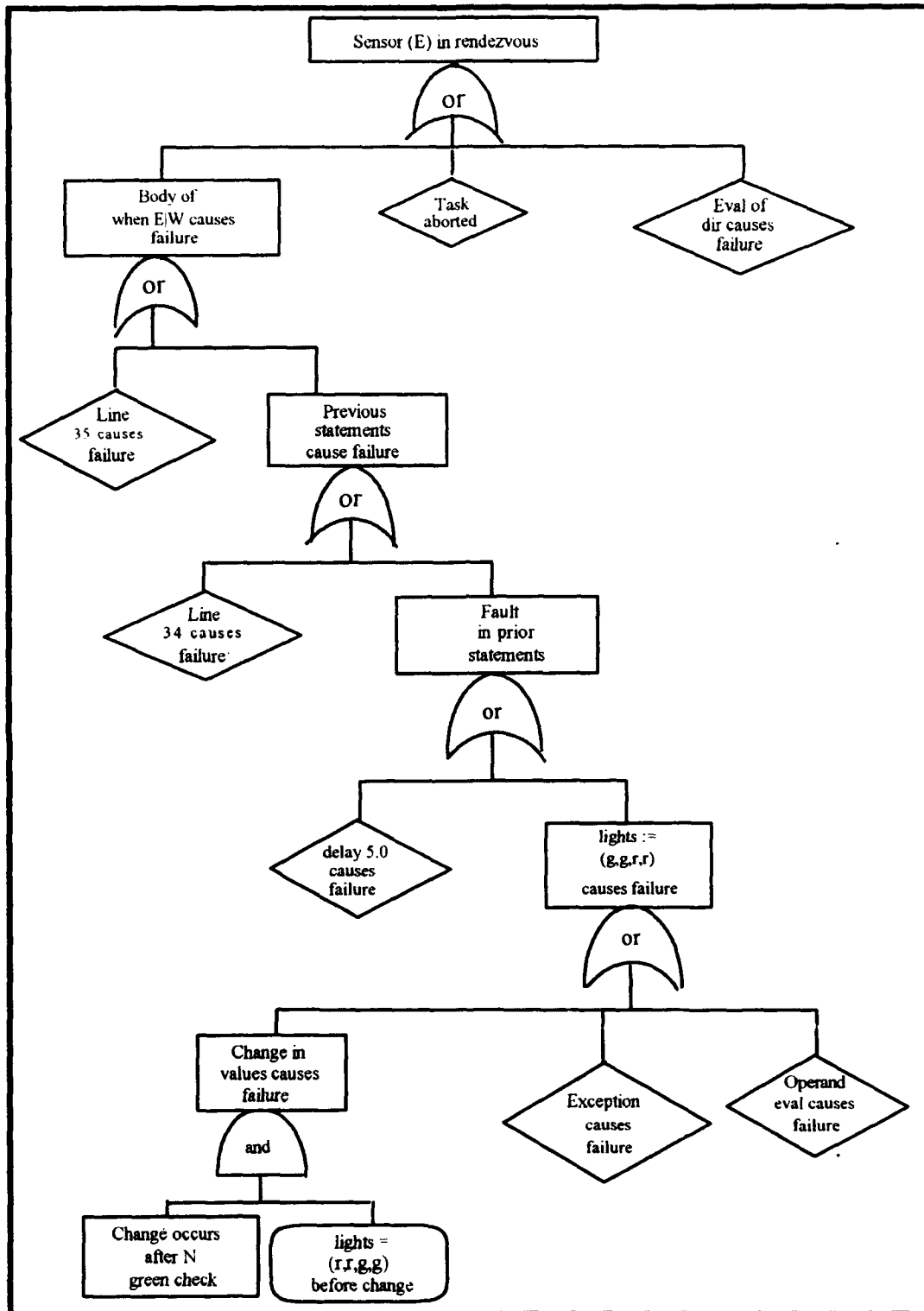


Figure 2-7 Sensor(East) in Rendezvous

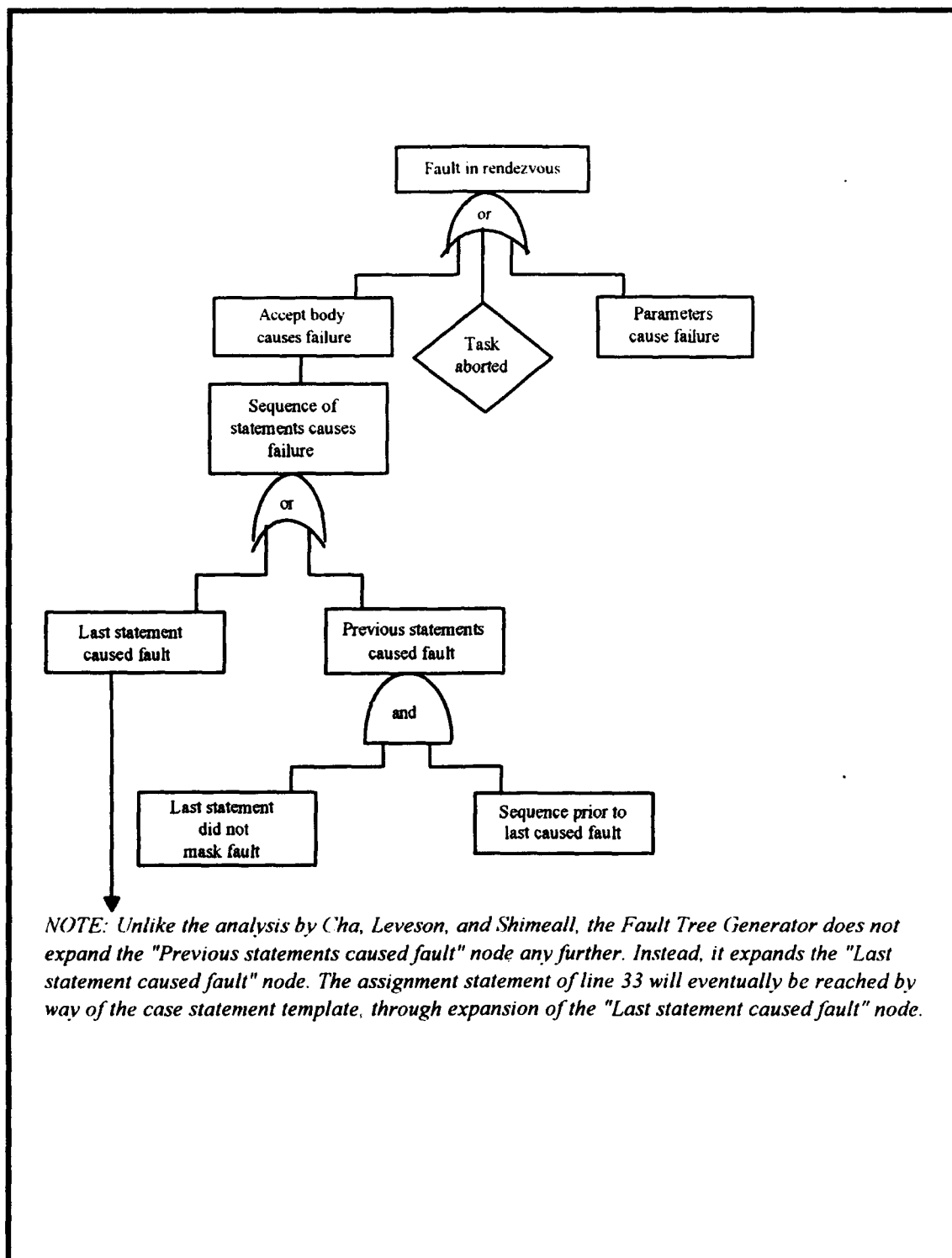


Figure 2-8 Tool Generated Fault Tree for TRAFFIC.A

III. SOFTWARE ANALYSIS

A. SYSTEM BACKGROUND

The Parallel Processor Based Small Tactical System Simulation was developed as part of The Parallel Command and Decision System (PARCDS) laboratory at Naval Postgraduate School. The laboratory was established in the early 1980's to support research for the Navy's AEGIS combat system. The Small Tactical System was modeled using a network of transputers, and implemented using the Ada programming language in conjunction with the Alsys-Ada Compiler. Produced by Alsys Limited, United Kingdom, this was the first compiler capable of supporting multiprocessor programming in Ada [Ref. 21].

The Aegis system, originally designed for the Ticonderoga class (CG-47) guided missile cruiser, consists of a three dimensional Phase Array Radar AN/SPY-1, the Command and Decision system (the four processor AN/UYK-7 computer system), and a weapons control system. Early research at the PARCDS laboratory involved tightly connected single-processor systems modeling parallel processing. The objective of the Small Tactical System project was to investigate the possibility of replacing the old standard Navy's computers for the Aegis real-time combat system aboard Naval ships with a network of transputers in order to reduce the reaction time of the Command and Decision System [Ref. 21].

B. SYSTEM DESCRIPTION

The Small Tactical System functions as follows. After target detection, a decision is made regarding target attack. If the decision to attack is made, then the target is tracked. A future position is estimated, and a weapon is launched to intercept the target at a predicted intercept point. These tasks are divided among three functional subsystems, a Target Tracker Subsystem, a Target Prediction Subsystem, and a Ballistic Interception Subsystem. These three subsystems are implemented using a network of five transputers (see Figure 3-1):

- T_0 is the host transputer which performs the Human Interaction.
- T_1 performs as the Target Tracker Subsystem.
- T_2 performs as the Target Prediction Subsystem.
- T_3 performs as the Ballistic Interception Subsystem.
- T_4 is a hot spare to make the system Fault Tolerant.

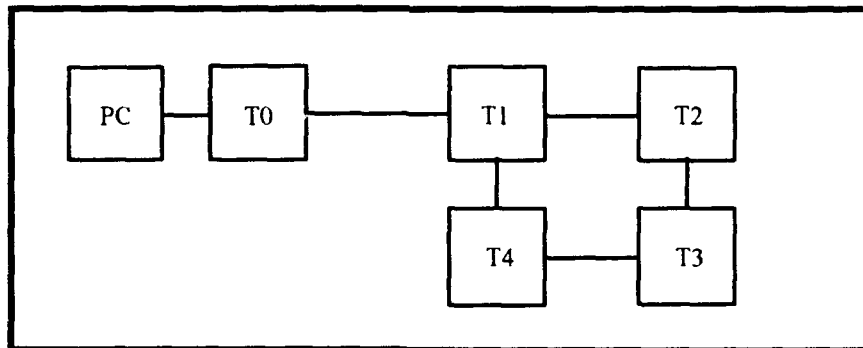


Figure 3-1 The designed Small Tactical System

The Target Tracker Subsystem code is the subject of analysis in this thesis.

C. GENERAL SOFTWARE DESCRIPTION

Radar echo information contains range and bearing to a target of interest. Further processing of this echo information results in target motion data which may be used in weapons attack. The tracked data simulation should simulate this same echo information.

The Ada code in Appendix D outputs simulated tracked positions of the target in three dimensions. The Prediction Subsystem requires this three dimensional data for the last seven position values be sent at one second intervals. The program assumes that the target approaches with some acceleration until it reaches its maximum speed, after which, all velocities remain constant. The program provides a textual output of the positions and velocities of the target every second. In the actual implementation, the output data would instead be sent to the Prediction Subsystem using transputer communication links.

D. SOFTWARE ANALYSIS

Remember that the basic procedure in FTA involves the assumption that the hazard has occurred, and now we must work backwards to determine its set of possible causes.

Also, to summarize, the fault tree templates are based on the following assumptions [Ref.

10]:

- The Ada program being analyzed is free from any syntax errors.
- The implementation of the underlying virtual machines are perfect.
- Templates currently refer to faults made in the program body - faulty declarations are not analyzed.
- Statements such as **GOTO**, are difficult to analyze by a backward trace, and thus are not analyzed.

The hazard event to be analyzed will be *missile launched at incorrect target*.

1. Localization

Refer to Figure 3-2. The main objective of the Ballistic Interception Subsystem is to compute intercept time. It accomplishes this after first computing the intercept distance.

Two of the assumptions of the Small Tactical System simulation are 1) a constant ammo speed, and 2) a straight line bullet trajectory. The intercept time is computed by dividing the intercept distance by the ammo speed. Based on the above assumptions, a missile launch at an incorrect target would result from an error in the computed intercept distance. [Ref. 21]

The Ballistic Interception Subsystem computes the intercept distance using predicted path lines input from the Target Prediction Subsystem. These path lines are computed from position and velocity vector inputs from the Target Tracking Subsystem. The Target Prediction Subsystem predicts the future position of the target from these vector inputs using the Least Square Orthogonal Polynomial. Formulas used in procedures and functions in both the Ballistic Interception and Target Prediction Subsystems (i.e., Simpson's Rule, Least Square Orthogonal Polynomial) are assumed to be error free. We therefore narrow the source of the error to the Target Tracker Subsystem. The code for the Target Tracker Subsystem is contained within the procedure PROJECT.⁶

⁶In the original document [Ref. 19], the procedure name was PROJ vice PROJECT.

Refer to Figure 3-3 and Appendix D. Examining the body of the procedure PROJECT, we see that the Target Tracker Subsystem will output incorrect vector quantities (**POSITION, VELOCITY**) if:

- the procedure GET_VELOCITIES retrieves incorrect values, or
- the velocity vector itself is in error. (**ACCELEROMETER.VEL**)

The assignment statements containing the call to the function SIMPSON are discounted, going along with our assumption that all formulas used are error free. We now examine the code pertaining to the Target Tracker Subsystem.

2. Isolation

The tool produces several output files for the analyst in addition to the terminal output. It is recommended that the user redirect the terminal output to a file due to its length. The file NEW_FTE contains the fault tree for all sections of code from the source file with the exception of task bodies. The tool generates a separate FTE file for each task body in the source code and names them sequentially, starting with TASK_BODYA. The terminal output contains the following information:

- Source listing with line numbers.
- **Literal Tree Output**, containing Parent node, Node, Parent Gate, and Fault for Node
- **FTE File Output**, containing Node, and Fault for Node
- Listing of procedures, functions, and tasks, and their root faults
- List of exceptions
- List of possible interleavings (rendezvous')

As a start, the tool tells us that there are a total of seven procedures, functions, or tasks:

- "+"
- "-"
- SIMPSON
- INITIALIZE_VELOCITY
- GET_VELOCITIES

- ACCELEROMETER and
- PROJECT.

There is also one rendezvous; "START". Examination of the code reveals that "+" and "-" are overloaded functions. Simpson's Rule is used in the function SIMPSON to compute the position vector, and PROJECT is the main procedure. INITIALIZE_VELOCITY, GET_VELOCITIES, both procedures, and task ACCELEROMETER are contained in the package ATOD.

The overloaded functions "+" and "-", and the function SIMPSON perform only the stated calculations, and as such may be analyzed in the same manner as discussed in previous works [Ref. 10 & 18]. INITIALIZE_VELOCITY is used set the initial velocity vector. Since this procedure is used to assign constant values (and we must assume that these values are both realistic and correct), it can be disregarded as a contributor to the hazard event. GET_VELOCITIES assigns to the variable NEW_VEL, the contents of the array VEL, which contains the last five velocity vectors. The contents of this array are generated within the body of task ACCELEROMETER. Therefore, we must examine the body of the task.

As with the traffic light example, we have again demonstrated that SFTA is very human-oriented. Automated tools are being developed primarily to aid in the analysis, not to replace the analyst.

The rendezvous START is null, and as such, can be ignored. Since the variable under examination (the array VEL) is local to the package ATOD, and its manipulation

within the task ACCELEROMETER is being analyzed, the flagged identifiers listing (printed out as each task body is parsed) can be ignored.

Looking at the code within task body ACCELEROMETER, we find that there are only two lines that are of concern, lines 128 and 130. As in the procedure INITIALIZE_VELOCITY, the assignment statement in line 130 can be ignored because it assigns a constant value to VEL(VELOCITIES'LAST). Now let's look at the surrounding code:

```
125 loop
126     delay DISP_TIME - SECONDS(CLOCK);
127     for I in VELOCITIES'FIRST .. VELOCITIES'LAST - 1 loop
128         VEL(I) := VEL(I + 1);
129     end loop
130, 131     .....
132 end loop;
```

The parameterless function **CLOCK** is provided by the predefined package **CALENDAR**. It returns a value of type **TIME** representing the moment at which the function was invoked. The function *should* return a different value each time it is called.

It can be seen that any adjustment or correction to the *system clock* (or frequency reference) during target tracking will result in the assignment of incorrect velocities for that time period, which will further result in a path prediction error. See Figure 3-4 for the tool generated fault tree for this section of code.

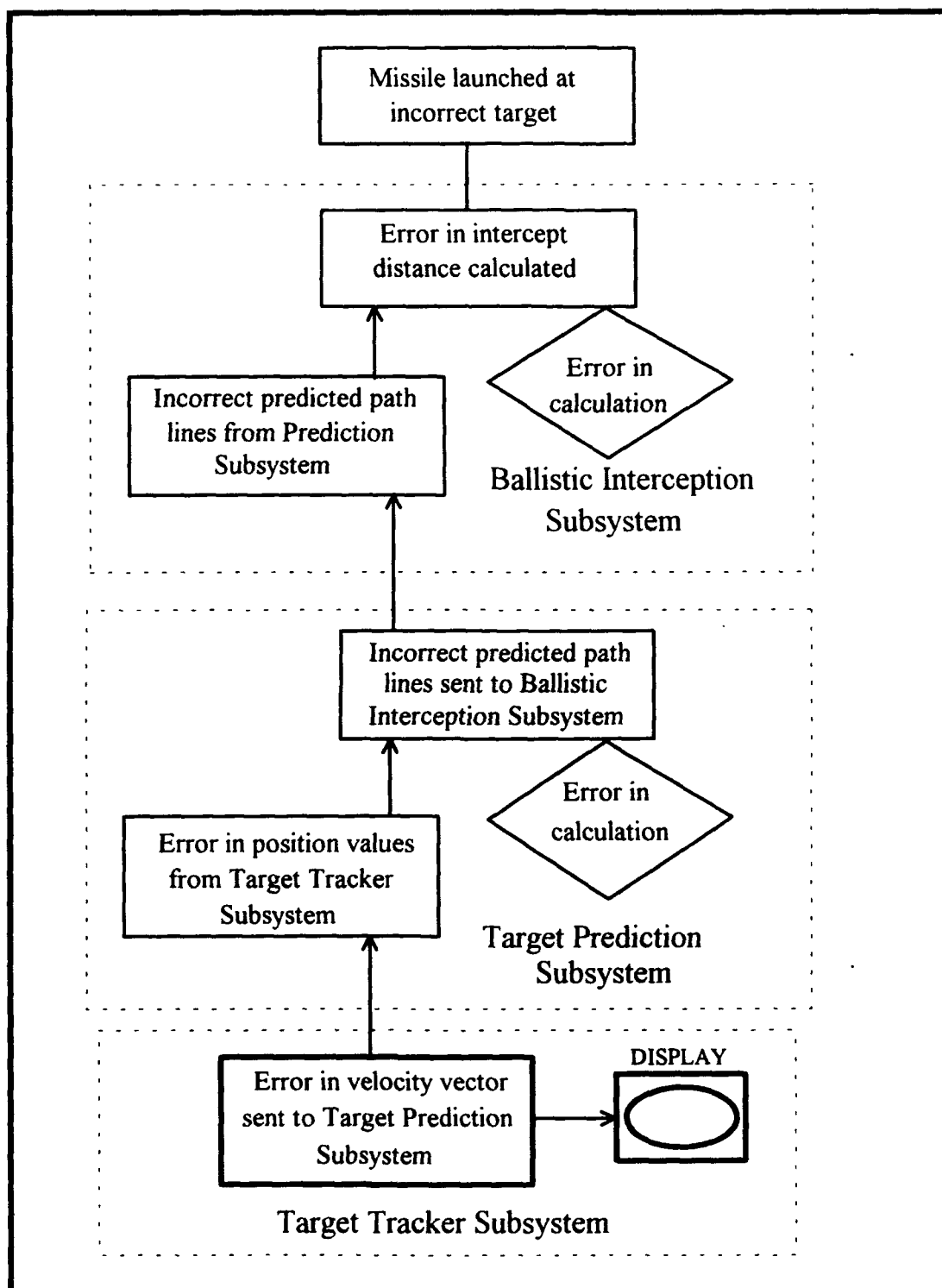


Figure 3-2 Top Level Fault Tree

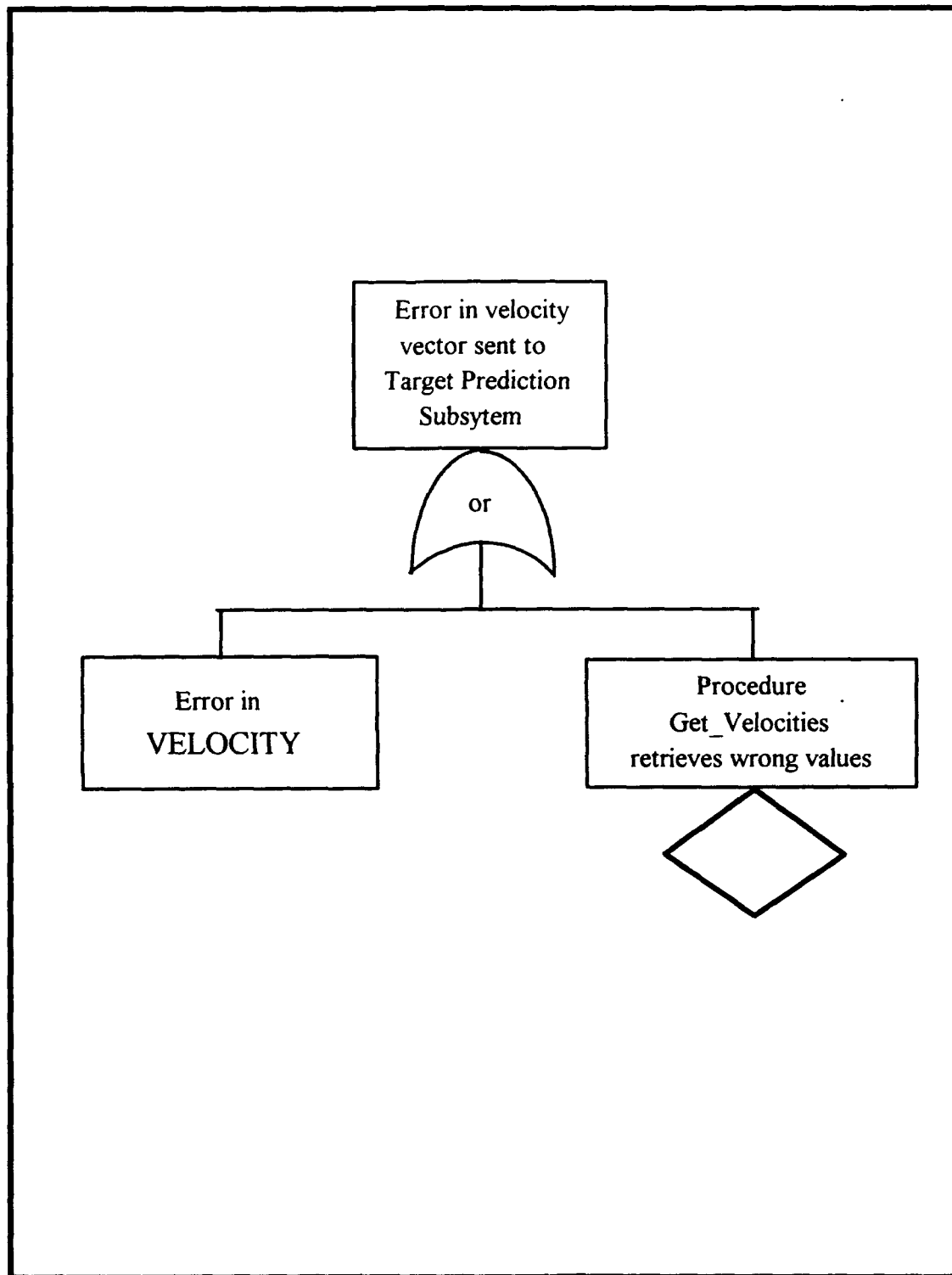


Figure 3-3 Error in Velocity Vector

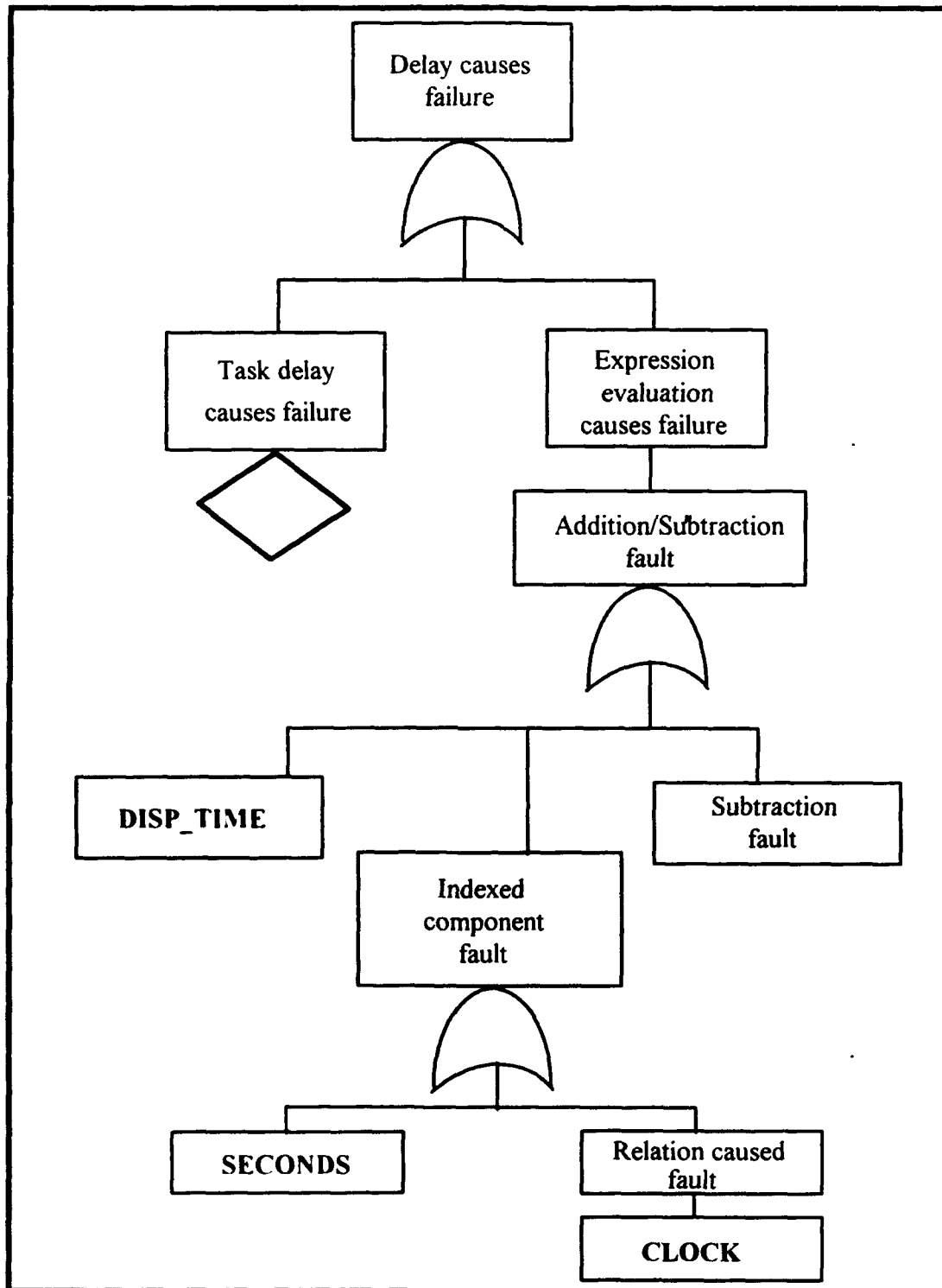


Figure 3-4 Tool Generated Fault Tree for Delay Statement Failure

IV. CONCLUSION

A. RESEARCH SUMMARY

In its chapter entitled, *"Survey of Software Verification and Validation Techniques"*, EWICS TC7 (European Workshop on Industrial Computer Systems, Technical Committee 7) suggests that the effectiveness of testing techniques depends not only on their proper application, but also on the procedures and standards followed in constructing the software [Ref. 24]. The aim of this thesis was to improve upon a prototype tool developed to automate the testing and analysis process, and demonstrate its application to a concurrent real-time system.

In this thesis, the addition of concurrency, and exception handling to the prototype tool developed by Ordonio [Ref. 18] was accomplished. The chosen avenue of providing the analyst with a separate fault tree for each task body is designed to focus his attention on this difficult aspect of Ada programming when it exists. Additionally, the tool was designed to flag concurrency *interaction* points (those having the capability to bring about unexpected results). These include shared variables, representation clauses, and rendezvous.

The exception handling template from Cha, Leveson, and Shimeall was added to the prototype, along with a modification of the original rendezvous template to include the

exception TASKING_ERROR. Like procedures, functions, and tasks, exception handlers found in the code are provided to the analyst in a listing.

In the traffic control system example, use of the fault tree generator provided immediate clues to areas of possible concern, and an ordered focus to the analysis. Use of automated tools is not meant to replace the analyst, but merely to remove some of the medial tasking.

The Aegis simulation analysis demonstrates that as complexity of the software increases, the analyst's knowledge of the system is instrumental in pruning the top level tree.

B. RECOMMENDATIONS

Who should use this tool? As has been demonstrated, the first obvious candidates are safety analysts. A major advantage gained in using this type of tool is that it allows the analyst to remove himself from concerns over syntax. With the tool, a proper balance has been maintained between human interaction and automation, greatly accelerating the human analyst's task.

Software *maintainers* need be especially concerned with concurrency. Unlike sequential programming, a hidden danger present in concurrent programming is that a program may depend subtly on the relative speeds of different processes. Additionally, certain errors may only arise dependant upon the timing of different tasks. When shared

variables are involved, concurrent task manipulations may be interleaved in unpredictable ways.

When software changes are made, analysis may be performed to detect new possible areas which may contribute to failure. The following statement is an excerpt from the investigation into the Therac-25 accidents:

It is clear from the AECL (Atomic Energy of Canada Limited) documentation on the modifications that the software allows concurrent access to shared memory, that there is no real synchronization aside from data stored in shared variables, and that the "test" and "set" for such variables are not indivisible operations. Race conditions resulting from this implementation of multitasking played an important part in the accidents." [Ref. 25]

Researchers may find the tool useful for three reasons:

- As an example of development of automated analysis tools of tasking software, getting around the state explosion of tasking by annotating the interaction points and letting the human determine the appropriate interactions to explore.
- As a basis of further research into analysis of concurrent software.
- As a basis for automating safety-critical software, integrating other techniques with fault tree analysis or producing more refined templates.

C. FUTURE RESEARCH

Software analysis of safety-critical software is presently evolutionary vice revolutionary. The problems and disadvantages of program analysis have so often been enumerated and explained.

- It is difficult to apply to larger programs.
- Some program constructs are difficult to analyze.
- It is difficult to analyze concurrent and real-time programs.
- The method can be time-consuming and error-prone if the analysis is performed manually.
- It gives no indication of totally missed parts. [Ref. 24]

It is crucial that this evolutionary process continue.

In its present state, the Automated Code Translation Tool does not actually provide a fault tree per se, but instead, an aggregate of the templates. An important next step would be to provide the analyst with a 'front end' to the tool which would allow him to prune the tree of non-contributing branches, and generate an actual fault tree for a selected *root fault* from selected templates. This front end could also allow the analyst to integrate into the fault tree, information provided by other automated tools such as P-NUT.

Gill [Ref. 14] believed that Petri nets and fault trees were limited when used individually, but when combined, could be used quite beneficially. A potential area of research might involve automation of Gill's technique, with follow-on research pursuing the combination of that resultant work and the fault tree generator.

With the pending release of Ada-9X, the present tool will have to be modified to accommodate the new grammar rules. This should prove to be a relatively simple task. Ada-9X will also provide several new features to the language. These include but are not limited to:

- Object-oriented programming with run-time polymorphism.
- Access types have been extended to allow an access value to designate a subprogram or an object declared by an object declaration.
- Additional support for interfacing to other languages. [Ref. 26].

The tool can also be used as a model for the development of automated tools for other languages. C/C++ for example, have a process-based rather than thread based model of concurrency. Also, exception handling is dealt with much differently in C (and proposed for C++), in that the language allows for the resumption of processing at the point the exception was raised.

APPENDIX A: GENERAL FIGURES

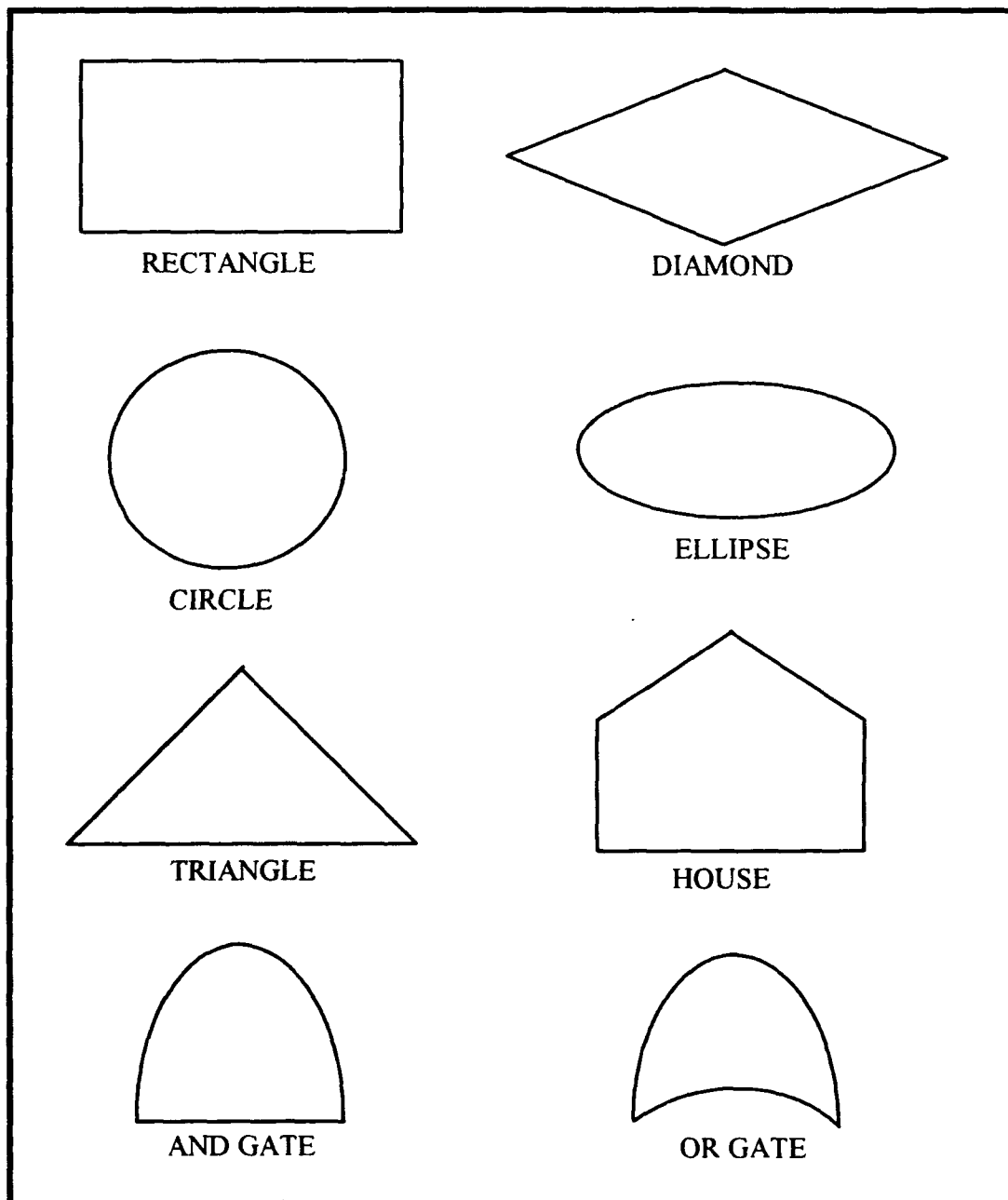
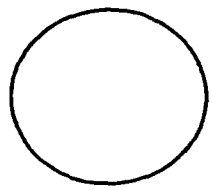


Figure A-1 Common Software Fault Tree Symbols

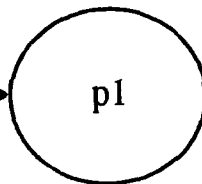


Circle represents a place

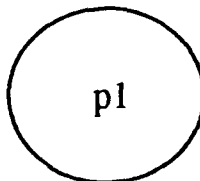


Bar represents a transition

Directed arcs connect the places and the transitions.



Transition t1 is an input to place p1



t1



Transition t1 is an output of place p1

Figure A-2 Petri Net Symbology

A transition is enabled if each of its input places has at least as many tokens in it as arcs from the places to the transition. A transition may fire only if it is enabled.

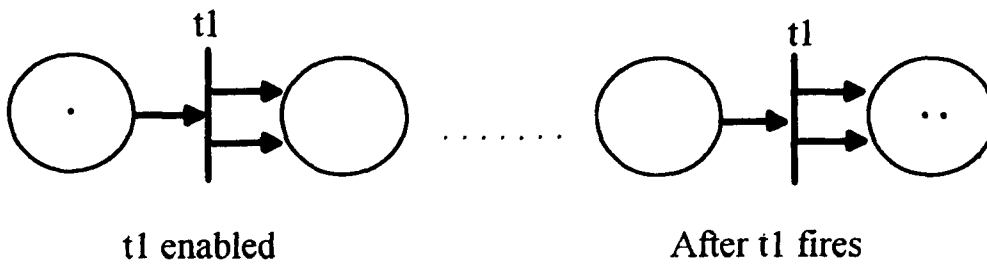
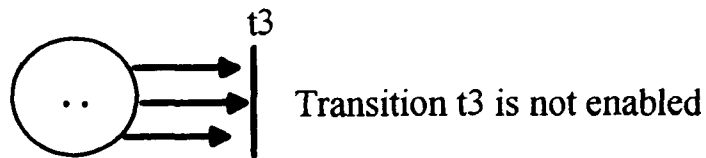
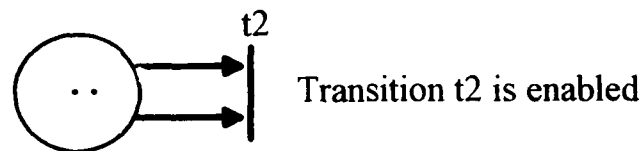
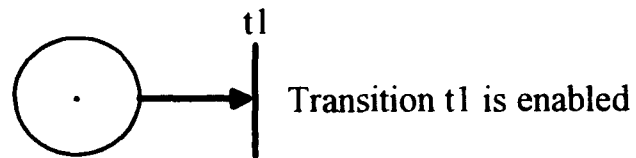
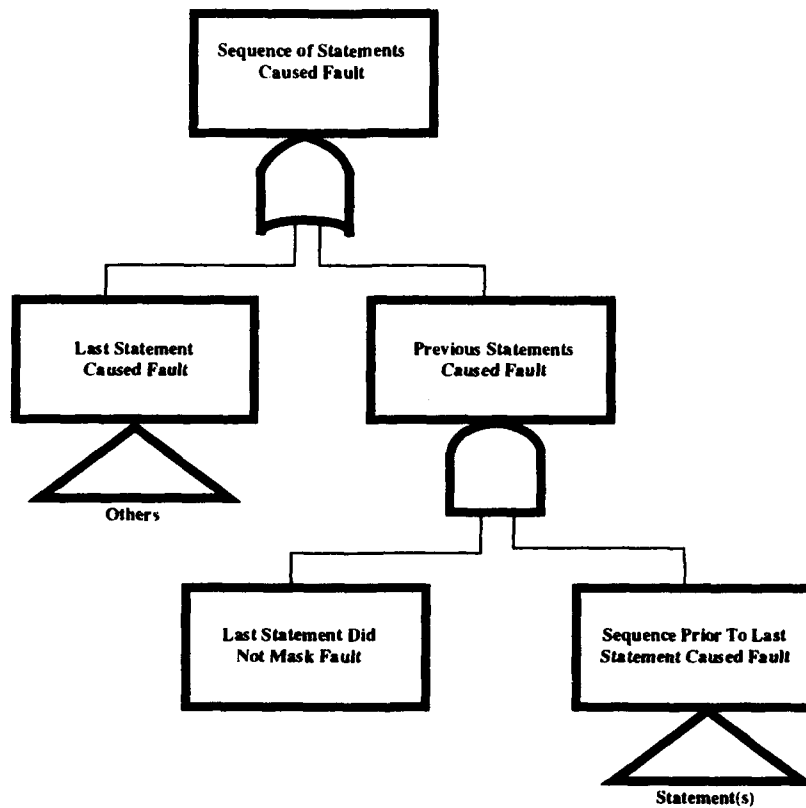


Figure A-3 Petri Net Execution

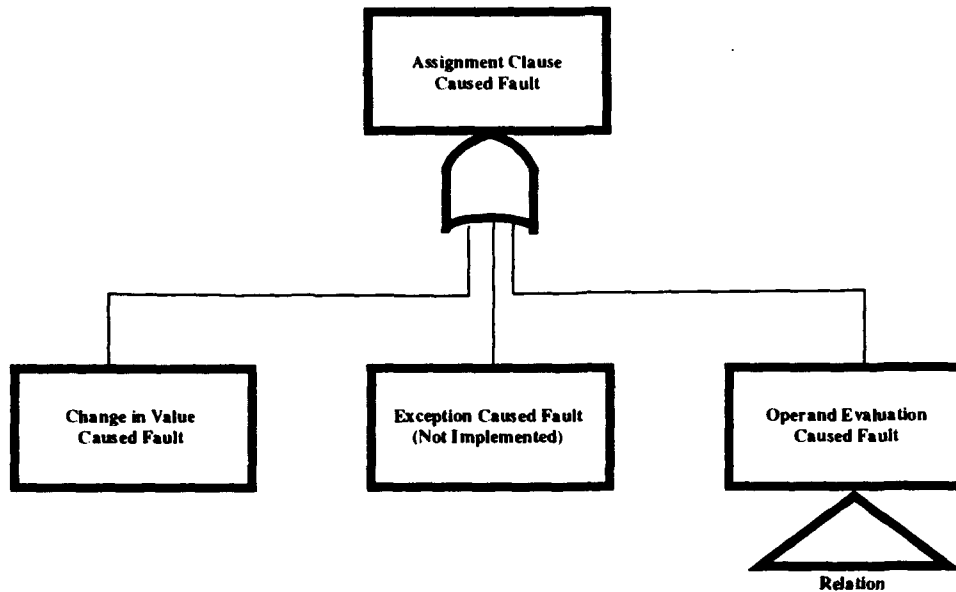
APPENDIX B: ADA STRUCTURE TEMPLATES

Sequence of Statements Template



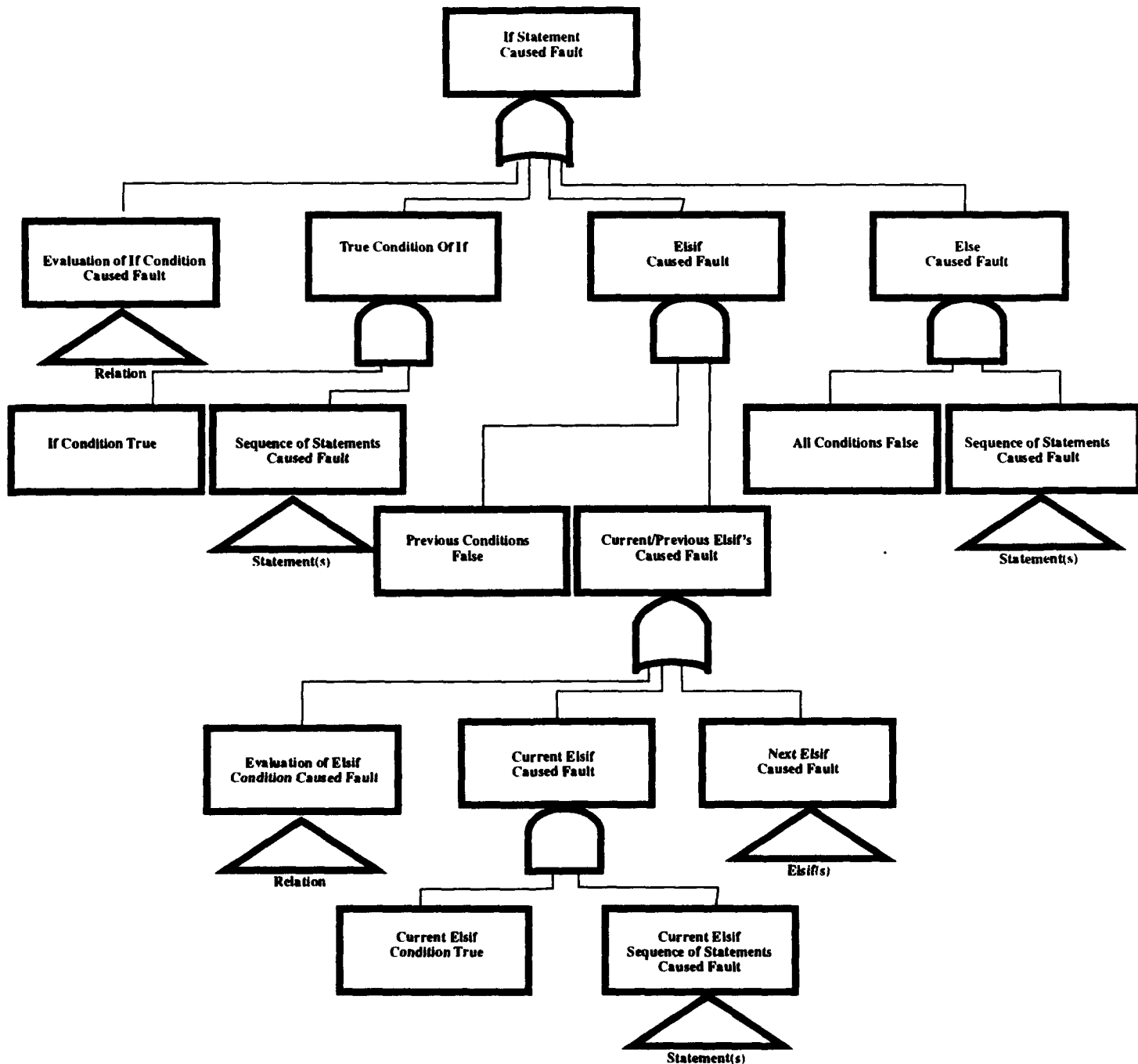
*** Sequence of statements template was not depicted in the works of Leveson, Cha, and Shimeall.**

Assignment Template



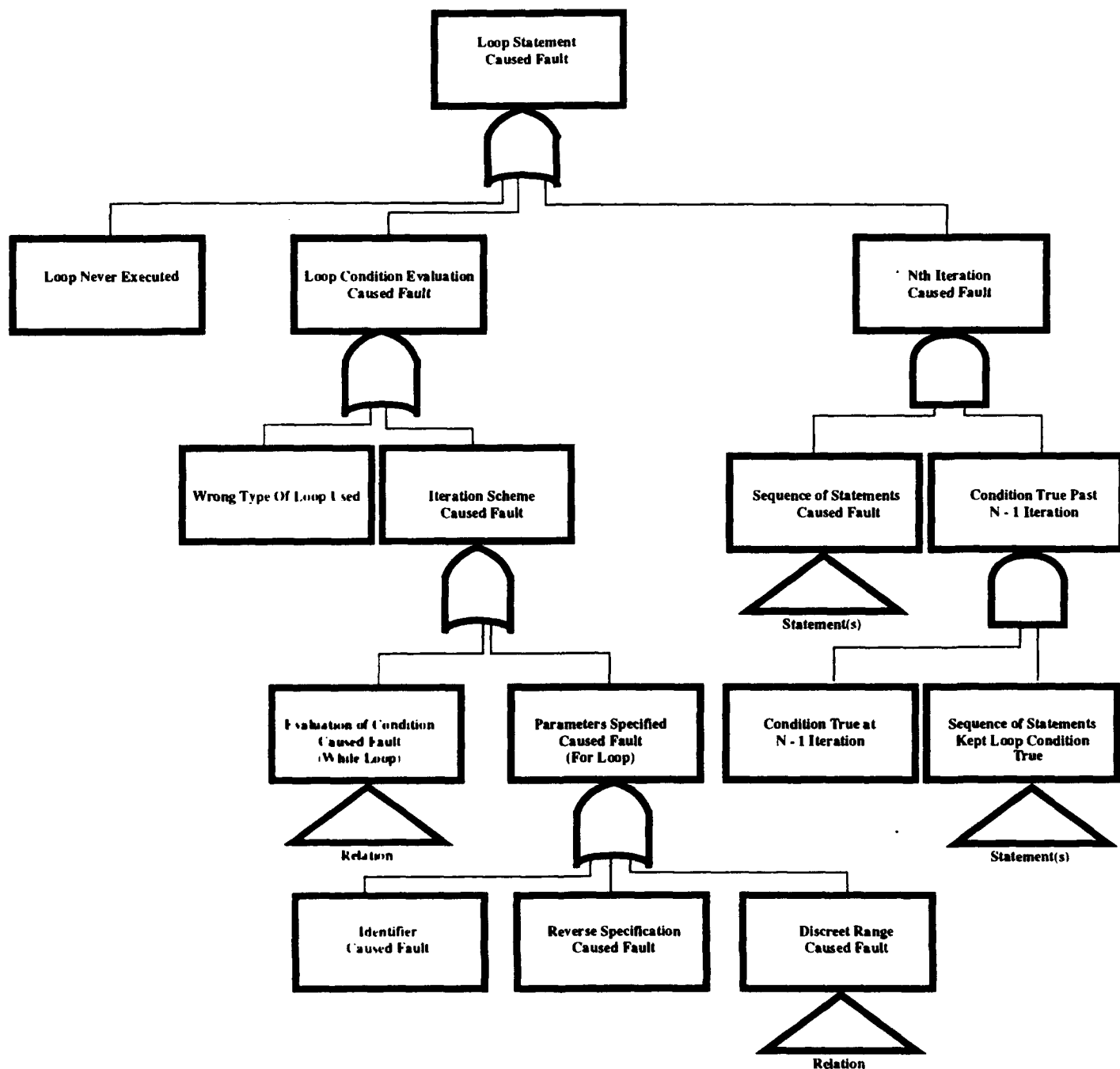
*** There is no difference between the tool generated Assignment Template and the Leveson, Cha, and Shimeall template.**

If Then Elsif Else Template



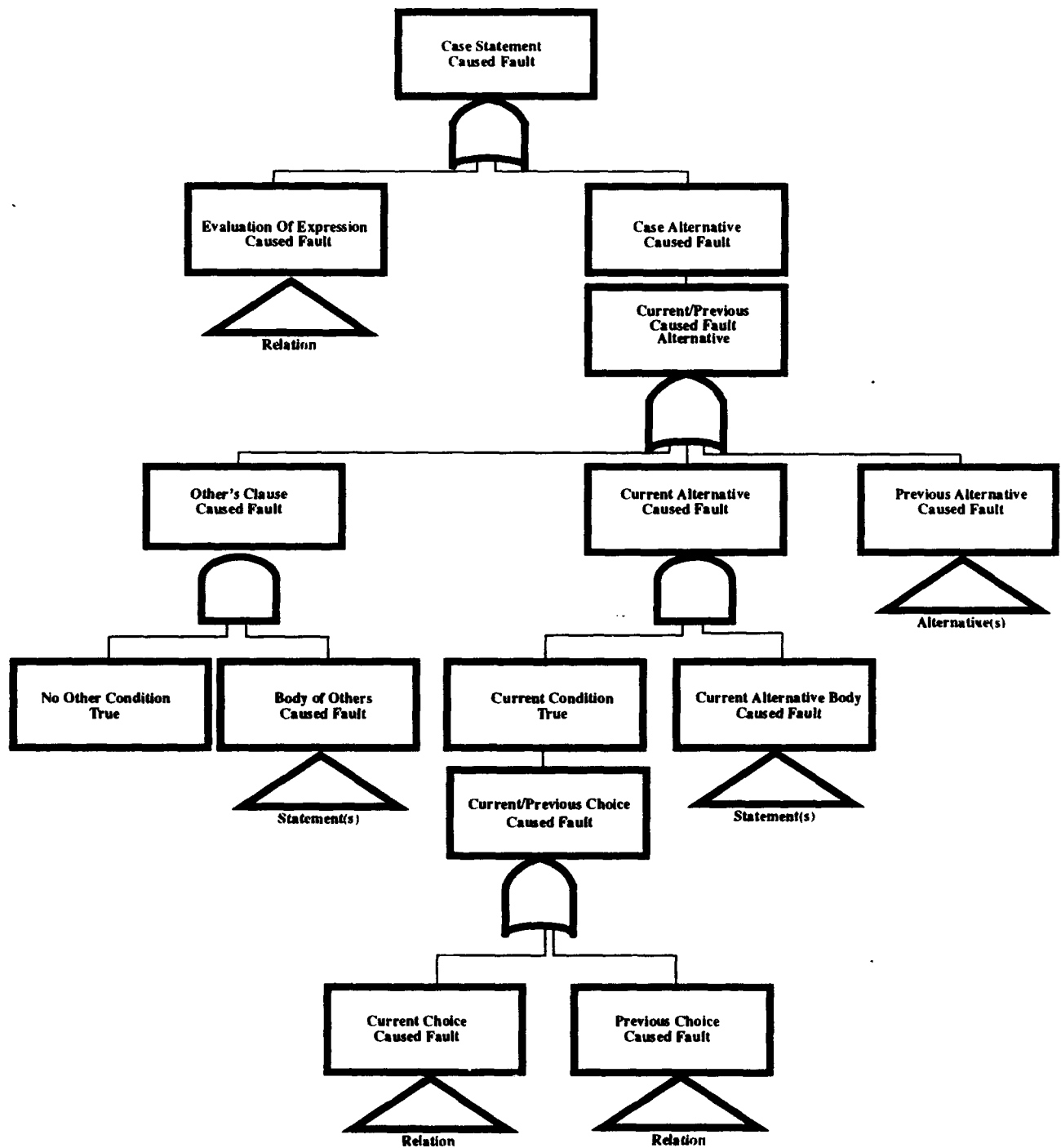
* The Leveson, Cha, and Shimeall template did not distinguish between the “if” associated nodes and the “elsif” associated nodes, but the tool generated template presented the two separately.

Loop Template



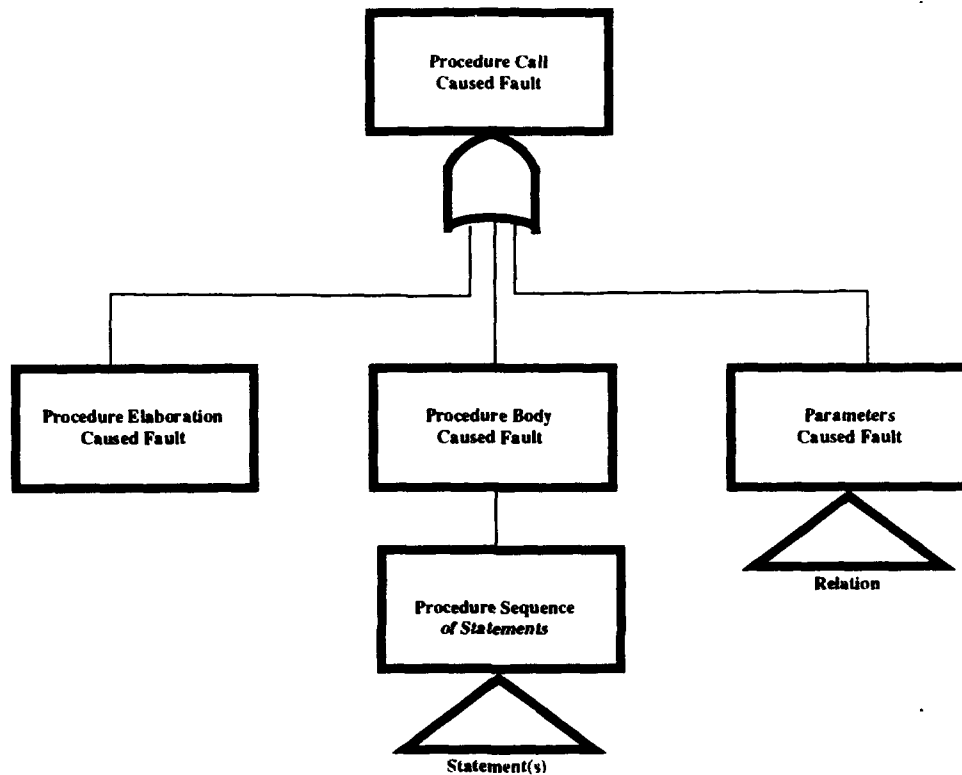
* The tool generated Loop Template distinguished between the different types of loops and expanded the associated “condition” nodes and the Leveson, Cha and Shimeall template did not.

Case Template



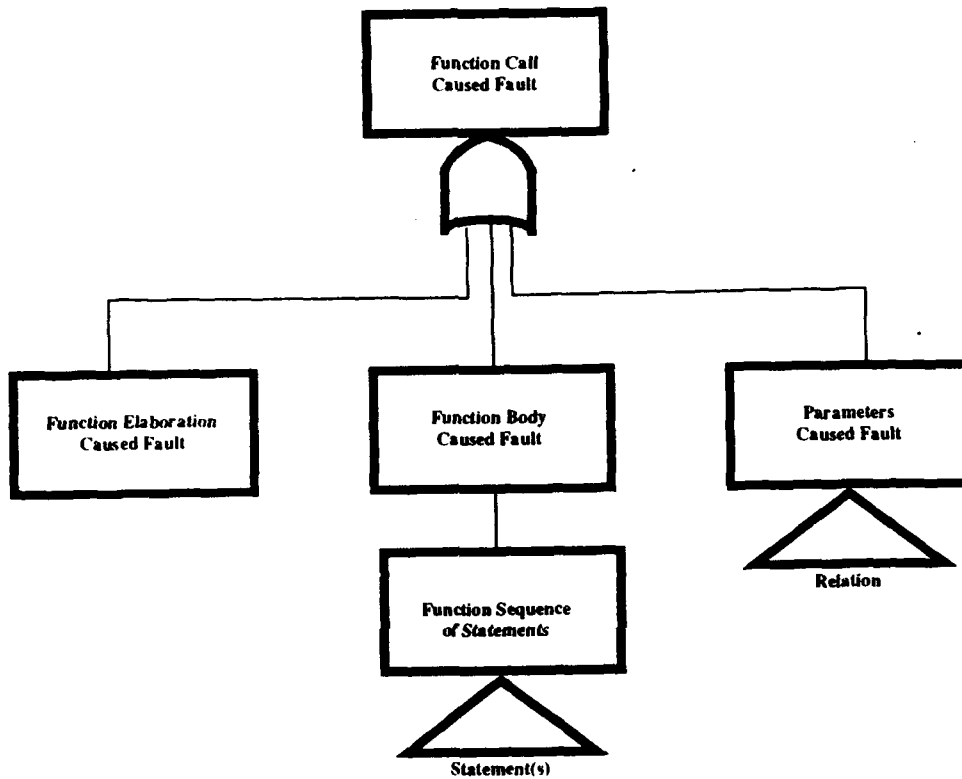
* The tool generated Case Template physically placed the “Other’s Clause Caused Fault” and associated nodes on a lower level then the Leveson, Cha, and Shimeall template, nevertheless, the two templates are equivalent.

Procedure Call Template



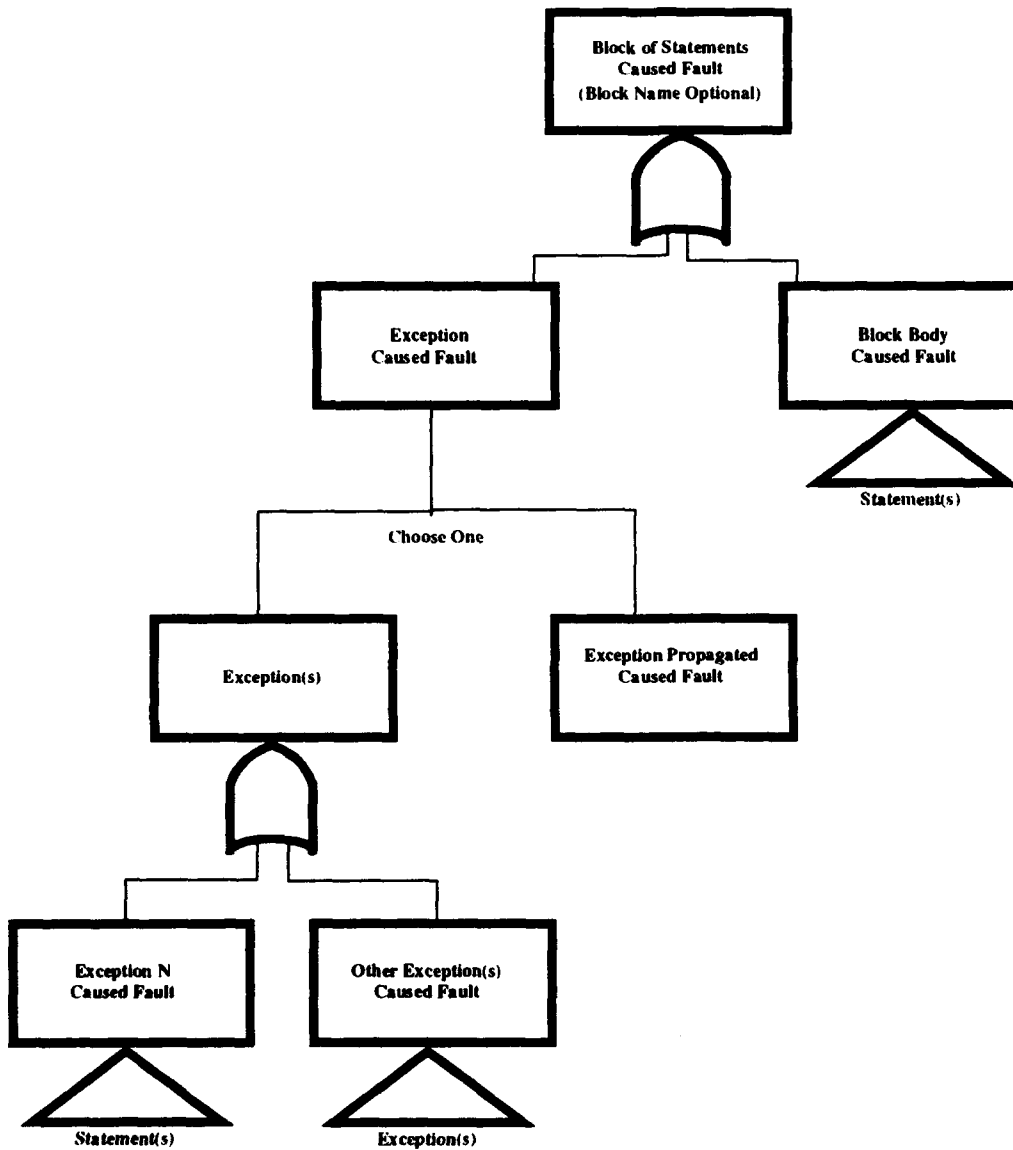
*** The tool generated Procedure Call Template added the node “Procedure Elaboration Caused Fault” to the template from Leveson, Cha, and Shimeall.**

Function Call Template



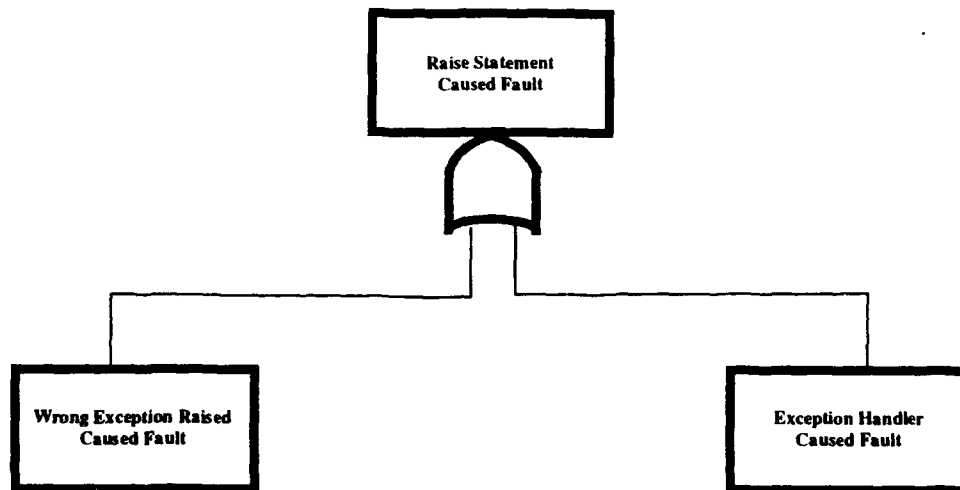
*** The Function Call Template was not depicted in the works of Leveson, Cha, and Shimeall.**

Block Statement Template



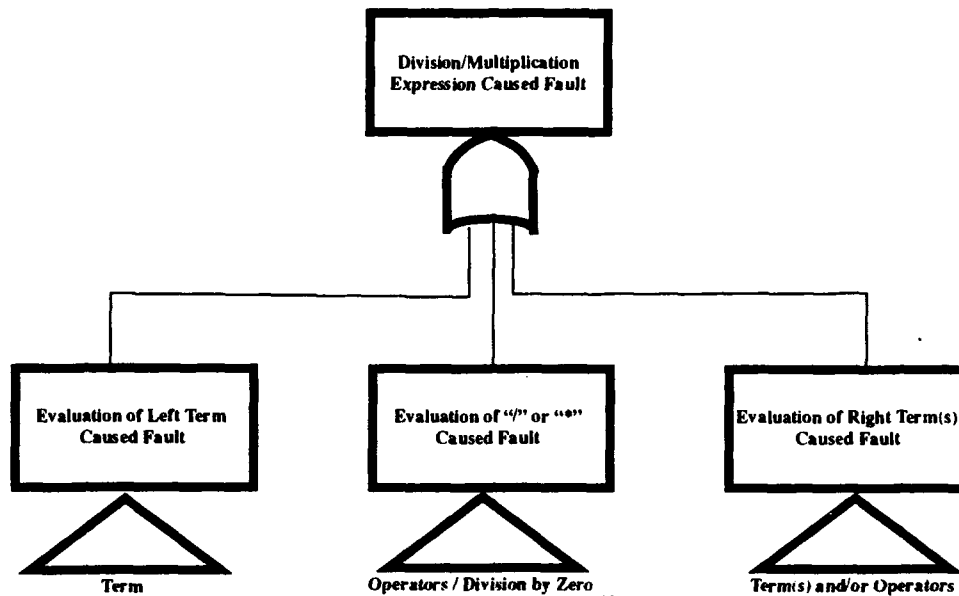
*** The tool generated Block Statement Template elaborated more on the "Exception Caused Fault" node then did the works of Leveson, Cha, and Shimeall.**

Raise Template



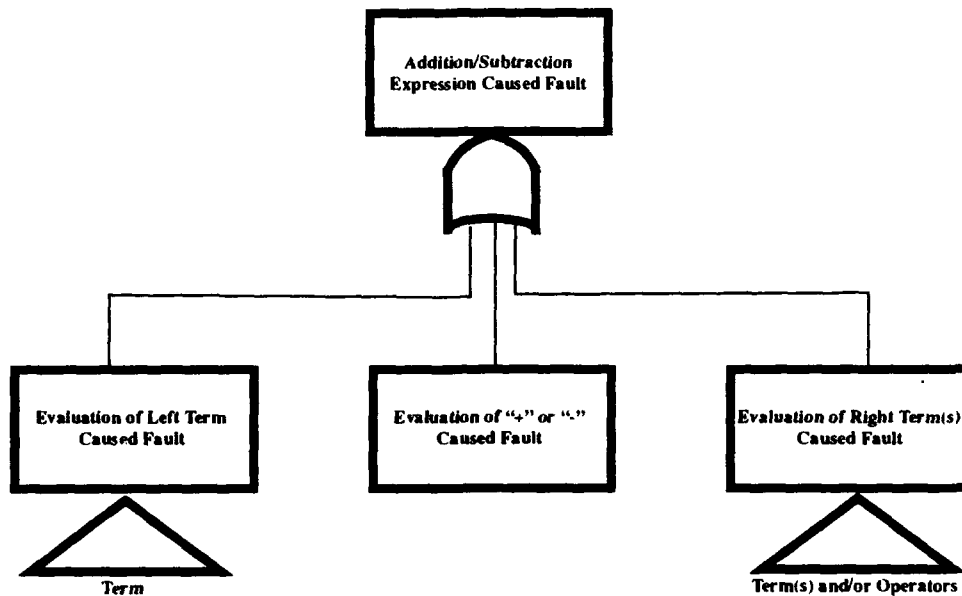
*** There was no difference in the Raise Template between the tool generated and the Leveson, Cha, and Shimeall templates.**

Division and Multiplication Template



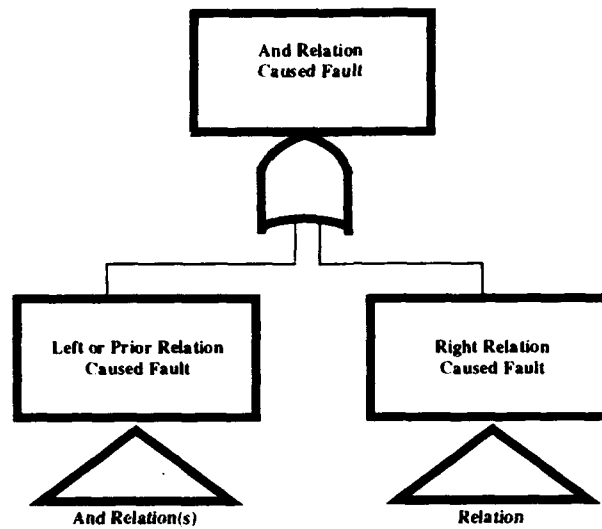
*** Division / Multiplication Template was not depicted in the works of Leveson, Cha, and Shimeall.**

Addition and Subtraction Template



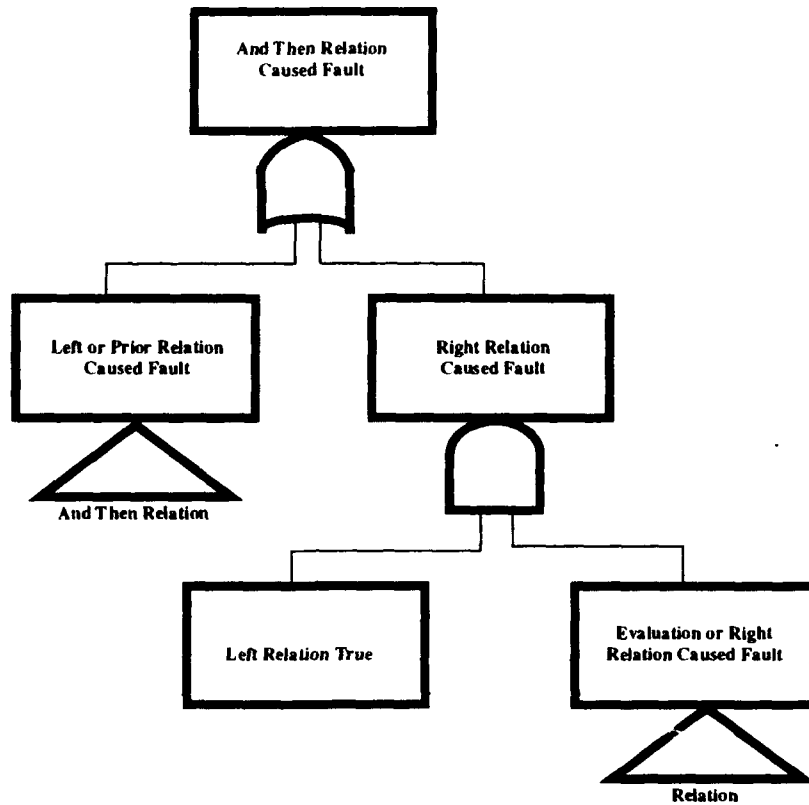
*** Addition / Subtraction Template was not depicted in the works of Leveson, Cha, and Shimeall.**

And Relation Template



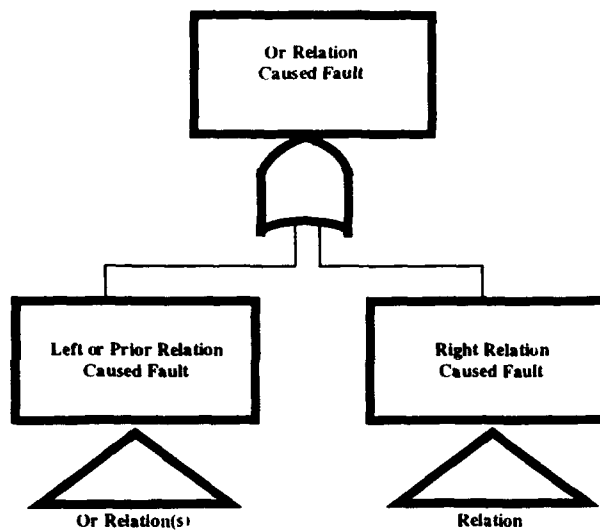
*** And Relation Template was not depicted in the works of Leveson, Cha, and Shimeall.**

And Then Relation Template



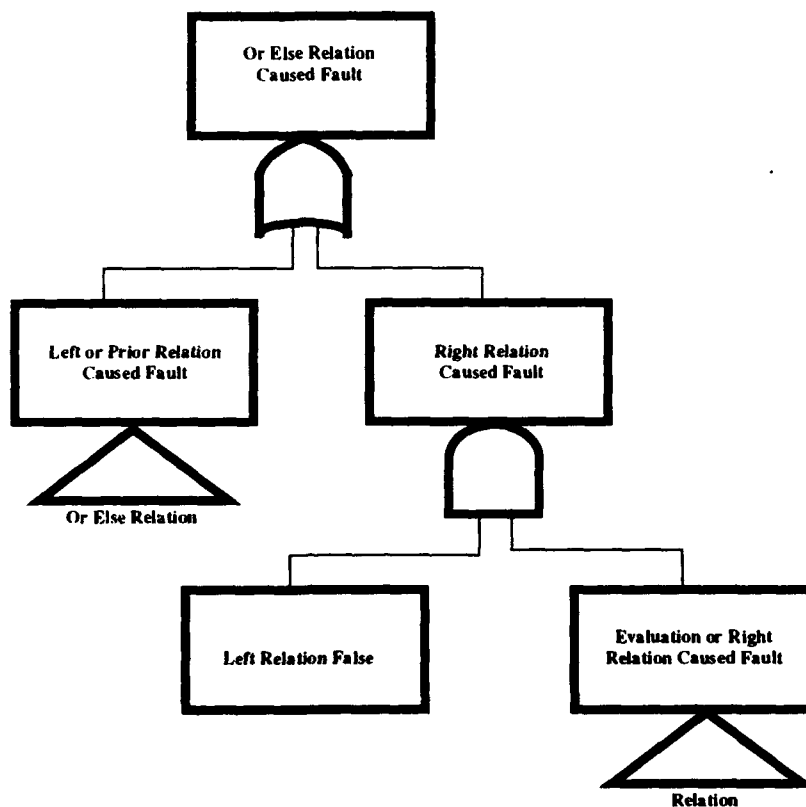
*** And Then Relation Template was not depicted in the works of Leveson, Cha, and Shimeall.**

Or Relation Template



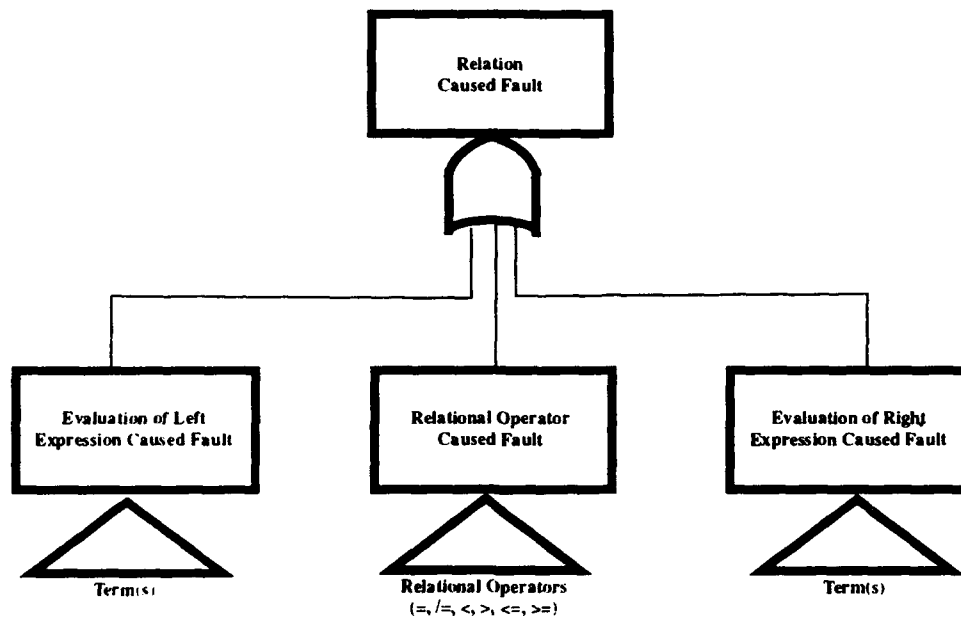
*** Or Relation Template was not depicted in the works of Leveson, Cha, and Shimeall.**

Or Else Relation Template



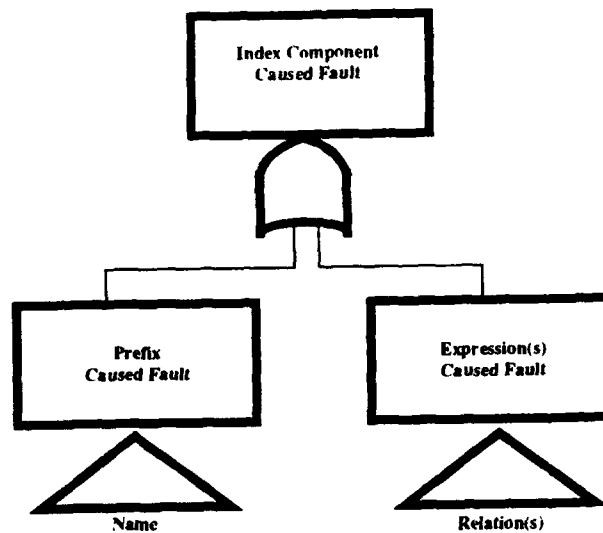
*** Or Else Relation Template was not depicted in the works of Leveson, Cha, and Shimeall.**

Relation Template



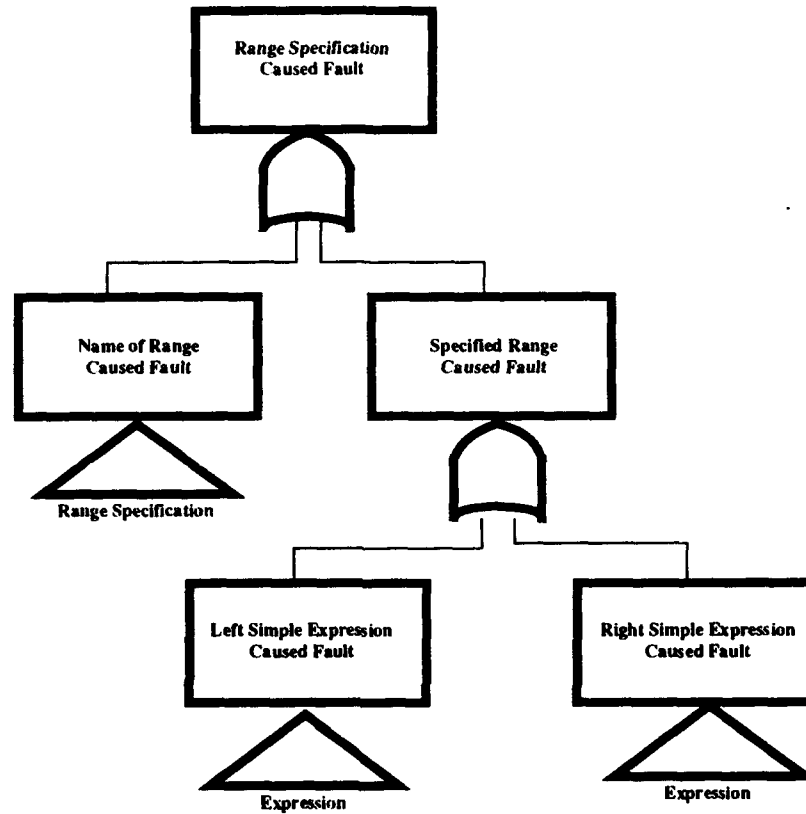
*** Relation Template was not depicted in the works of Leveson, Cha, and Shimeall.**

Index Component Template



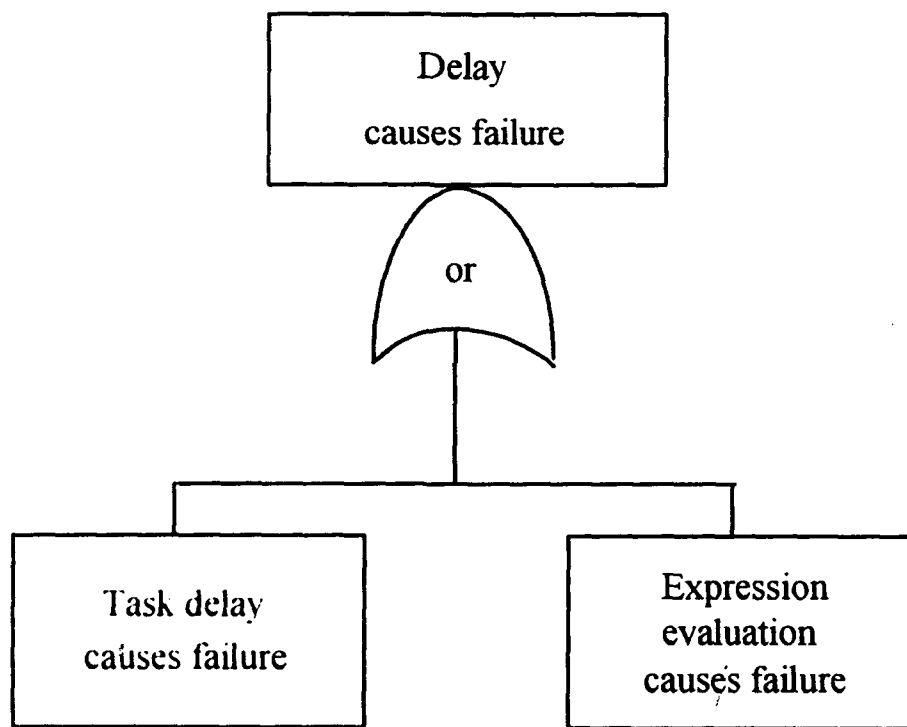
*** Index Component Template was not depicted in the works of Leveson, Cha, and Shimeall.**

Range Template

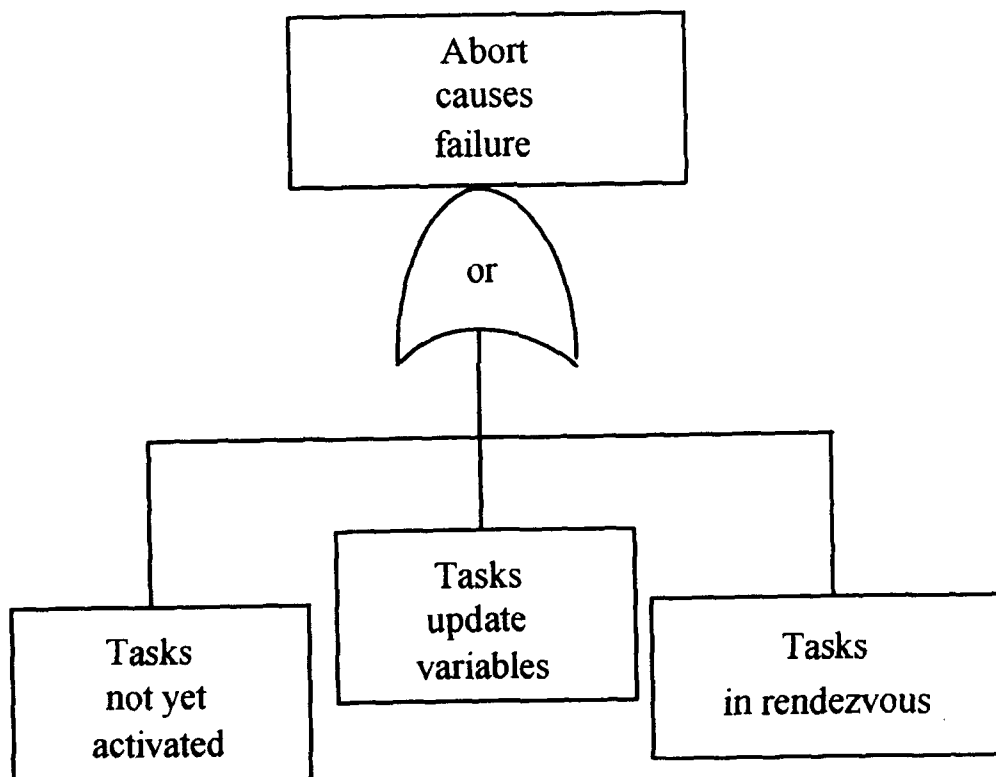


*** Range Template was not depicted in the works of Leveson, Cha, and Shimeall.**

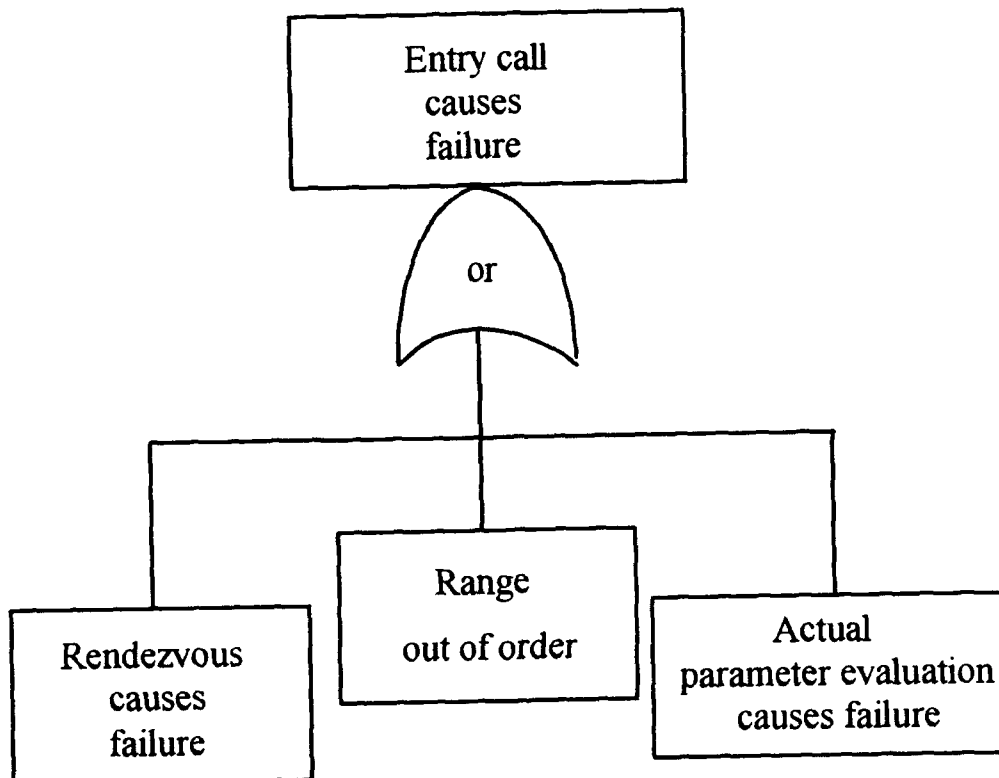
Delay Template



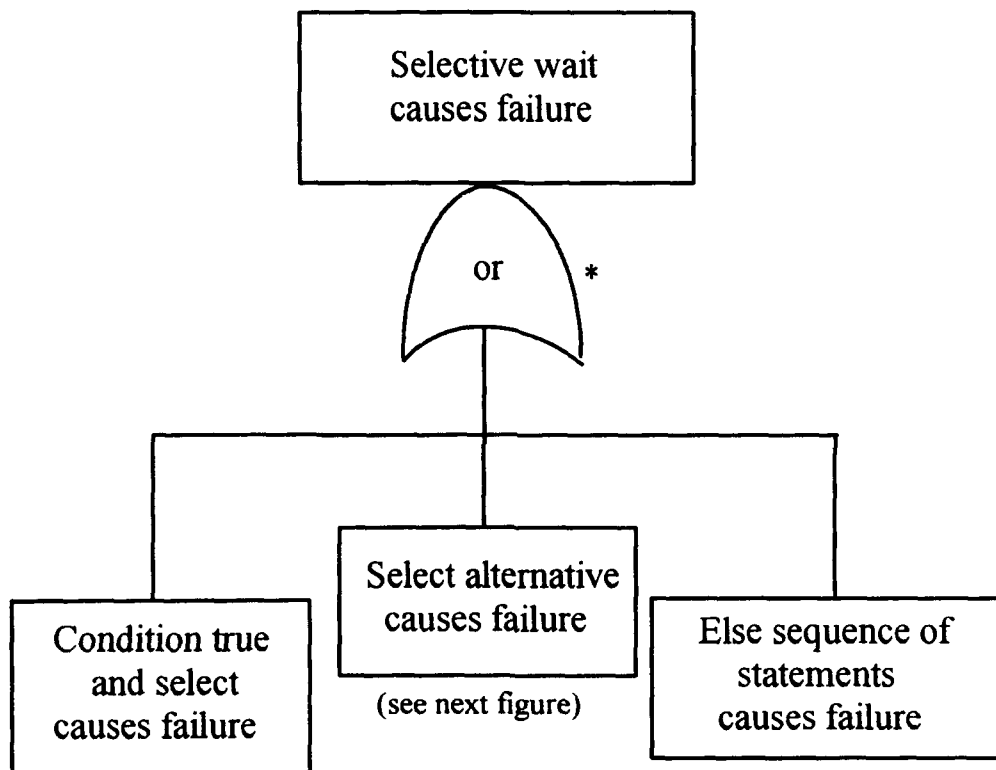
Abort Template



Entry Template

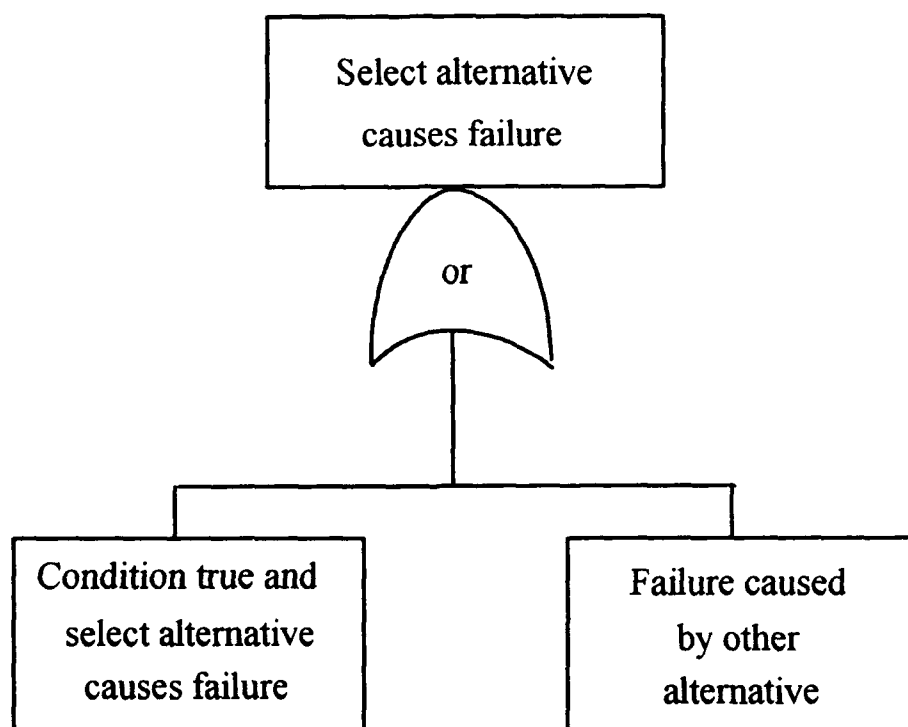


Selective Wait Template

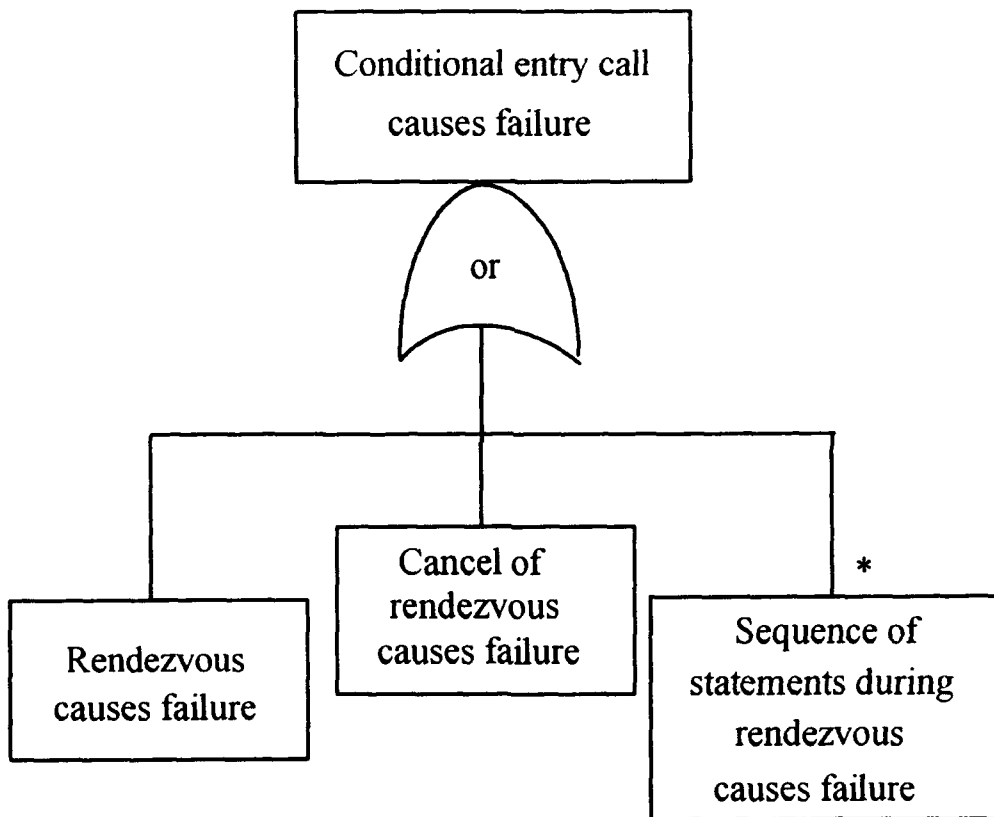


* The Cha, Leveson, and Shimeall select template contained and AND gate vice an OR gate at this location.

Select Alternative Template

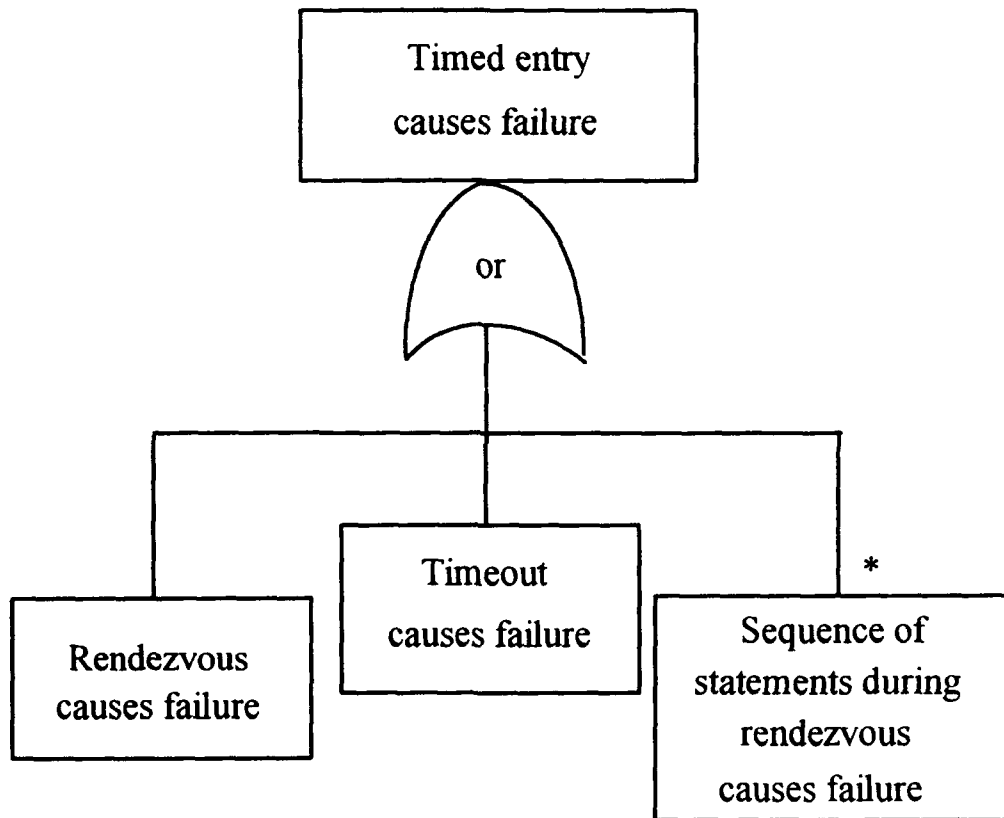


Conditional Entry Template



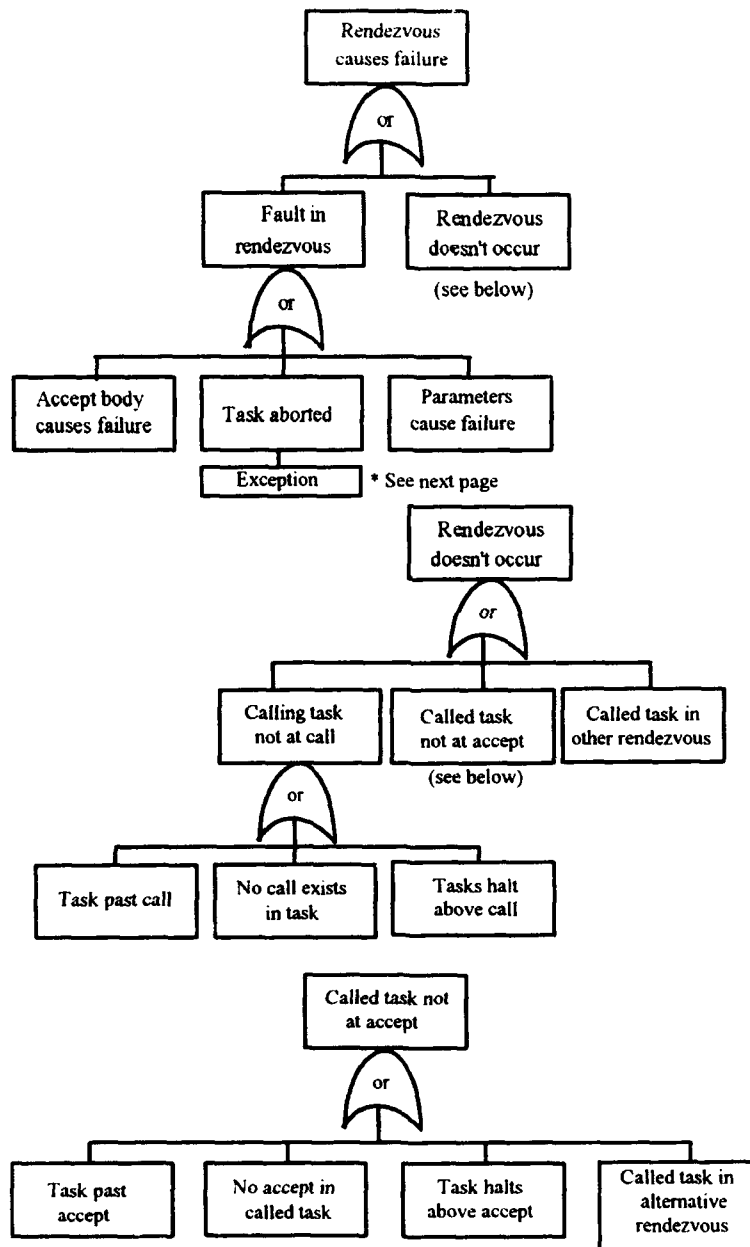
* This node was added to more accurately reflect the Ada grammar.

Timed Entry Template



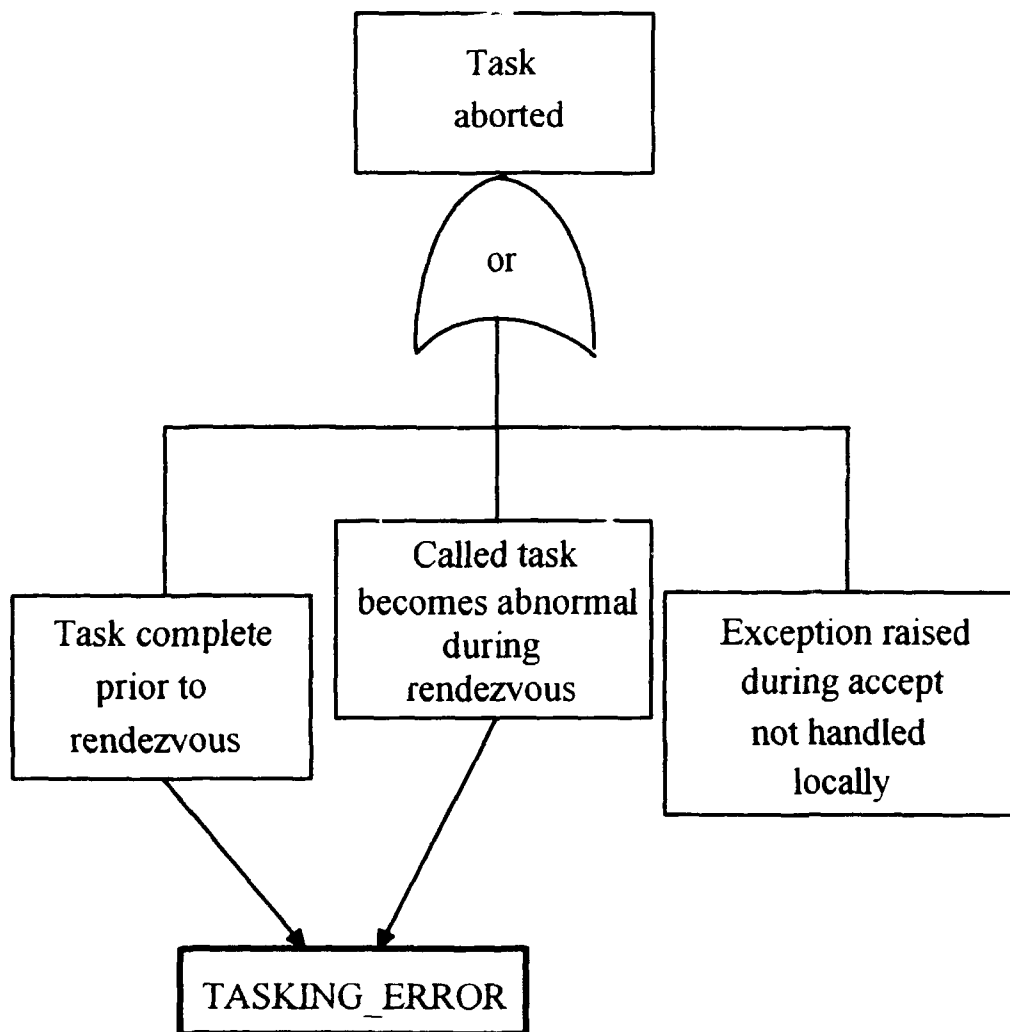
* This node was added to more accurately reflect the Ada grammar.

Rendezvous Template



Rendezvous Template (cont)

An exception can be raised in a task during a rendezvous or when attempting a rendezvous. The situations resulting in raising an exception are used to modify the original rendezvous template [Ref. 23]



APPENDIX C: FTE FILE INFORMATION

The fault tree files that can be read by the Fault Tree Editor (FTE), and that FTE writes have the following characteristics:

line 1:	label	string
line 2:	fault	string
line 3:	file	string
line 4:	start_line	integer
	end_line	integer
	x_coord	integer
	y_coord	integer
	type	integer
	gate	integer
	n_children	integer

Line 4 has seven fields separated by any whitespace except a newline. An FTE file consists of a series of the above four line groups, one group for each node. In addition the nodes must be ordered in "preorder" fashion to be understood by FTE. The best way to see this is to create a simple fault tree using FTE and then look at the file generated. An example of a single node follows:

Label:	"Top"
Fault:	"This program will always fail"
File:	"test.c"
start_line:	1
end_line:	55
x_coord:	200
y_coord:	300
type:	1
gate:	0
n_children:	0

The above node would look like:

```
Top
This program will always fail
test.c
1 55 200 300 1 0 0
```

in the FTE file saved.

APPENDIX D: PROJECT.A SOURCE LISTING

```
-- The original source code with'd and instantiated a generic_elementary_functions package
-- available in the Alsys-Ada Compilation System. The code as follows was
-- compiled and executed using both SunAda and Meridian Ada compilers without
-- error.
```

```
with text_io, calendar; -- with generic_elementary_functions
use text_io, calendar;
```

procedure PROJECT is

```
--package MATH_FUNCT is new GENERIC_ELEMENTARY_FUNCTIONS(FLOAT);
--use MATH_FUNCT;
```

```
package FLOAT_INOUT is new FLOAT_IO(FLOAT);
use FLOAT_INOUT;
```

```
--Type identifications
type COMPONENT is (X,Y,Z);
type VECTOR is array (COMPONENT) of FLOAT;
type VELOCITIES is array (0..4) of VECTOR;
```

```
DELTA_TIME : constant FLOAT := 0.25;
--delta time T=1/4 seconds
```

```
NO_TARGET : BOOLEAN := FALSE;
CURRENT   : constant INTEGER := 4;
LINT_SEC  : INTEGER;
INTERVAL  : DAY_DURATION := 1.0;
DISP_TIME : DAY_DURATION := 0.0;
POSITION  : VECTOR := (27000.0,22000.0,5000.0);
INT_VEL   : VECTOR := (230.0,180.0,25.0);
VELOCITY  : VELOCITIES;
```

function "+" (LEFT, RIGHT : in VECTOR) return VECTOR is

```
-- This function is written to handle VECTOR addition.
```

```
    TEMP : VECTOR;
begin
    TEMP(X) := LEFT(X) + RIGHT(X);
    TEMP(Y) := LEFT(Y) + RIGHT(Y);
    TEMP(Z) := LEFT(Z) + RIGHT(Z);
    return TEMP;
end "+";
```

function "-" (LEFT, RIGHT : in VECTOR) return VECTOR is

-- This function is written to handle VECTOR subtraction.

```
    TEMP : VECTOR;  
begin  
    TEMP(X) := LEFT(X) - RIGHT(X);  
    TEMP(Y) := LEFT(Y) - RIGHT(Y);  
    TEMP(Z) := LEFT(Z) - RIGHT(Z);  
    return TEMP;  
end "-";
```

function SIMPSON (XPOS : in VECTOR;
 VEL : in VELOCITIES) return VECTOR is

-- This function performs numeric integration using
-- Simpson's Rule and returns a position vector given
-- a set of sample VELOCITIES.

```
    T : VECTOR;  
    I : COMPONENT;  
    J : INTEGER;  
begin  
    for I in COMPONENT loop  
        T(I) := (VEL(VELOCITIES'FIRST) (I) + VEL (VELOCITIES'LAST) (I));  
        for J in VELOCITIES'FIRST + 1..VELOCITIES'LAST - 1 loop  
            if (J MOD 2) = 1 then  
                T(I) := T(I) + 4.0 * VEL(J) (I);  
            else  
                T(I) := T(I) + 2.0 * VEL(J) (I);  
            end if;  
        end loop;  
        T(I) := DELTA_TIME * T(I) / 3.0;  
    end loop;  
    return T;  
end SIMPSON;
```

package ATOD is

-- This package is used to maintain the velocity values for the
-- previous second at 1/4 second intervals (five values). It has
-- one function which returns an array of vectors. The task is
-- written to handle concurrent processing of ACCELEROMETER.

```
    procedure INITIALIZE_VELOCITY(FIRST_VEL : in VECTOR);  
    procedure GET_VELOCITIES(NEW_VEL : out VELOCITIES);  
    task ACCELEROMETER is  
        entry START;  
    end ACCELEROMETER;  
end ATOD;
```

```

package body ATOD is
  VEL : VELOCITIES := (others => (others => 0.0));

  procedure INITIALIZE_VELOCITY(FIRST_VEL : in VECTOR) is
    I : INTEGER;
  begin
    for I in VELOCITIES'RANGE loop
      VEL(I) := FIRST_VEL;
    end loop;
  end INITIALIZE_VELOCITY;

  procedure GET_VELOCITIES(NEW_VEL : out VELOCITIES) is
  begin
    NEW_VEL := VEL;
  end GET_VELOCITIES;

  task body ACCELEROMETER is
    use CALENDAR;
    INTERVAL : constant DURATION := 0.25;
    DISP_TIME : DURATION := 0.0;
    LINT_SEC : INTEGER := 0;
  begin
    accept START do
      null;
    end START;

    LINT_SEC := INTEGER(SECONDS(CLOCK));
    DISP_TIME := DURATION(LINT_SEC);
    while DISP_TIME < SECONDS(CLOCK) loop
      DISP_TIME := DISP_TIME + INTERVAL;
    end loop;
    loop
      delay DISP_TIME - SECONDS(CLOCK);
      for I in VELOCITIES'FIRST..VELOCITIES'LAST-1 loop
        VEL(I) := VEL(I+1);
      end loop;
      VEL(VELOCITIES'LAST) := VEL(VELOCITIES'LAST) +
        (0.012,0.0098,0.00275);
      exit when VEL(4)(X) > 700.0;
    end loop;
  end ACCELEROMETER;
end ATOD;

```

```

procedure PUT_POSITION_VELOCITY (XP : in VECTOR;
                                VEL : in VECTOR) is
begin
    SET_COL(2);
    PUT(XP(X), FORE => 6, AFT => 4, EXP => 0);
    PUT(" ");
    PUT(XP(Y), FORE => 6, AFT => 4, EXP => 0);
    PUT(" ");
    PUT(XP(Z), FORE => 6, AFT => 4, EXP => 0);
    PUT(" ");
    PUT(VEL(X), FORE => 6, AFT => 4, EXP => 0);
    PUT(" ");
    PUT(VEL(Y), FORE => 6, AFT => 4, EXP => 0);
    PUT(" ");
    PUT(VEL(Z), FORE => 6, AFT => 4, EXP => 0);
    NEW_LINE;
end PUT_POSITION_VELOCITY;

begin -- main program

    ATOD.INITIALIZE_VELOCITY(INT_VEL);
    LINT_SEC := INTEGER(SECONDS(CLOCK));
    DISP_TIME := DURATION(LINT_SEC) + 0.8;
    ATOD.ACCELEROMETER.START;
    while NO_TARGET = FALSE loop
        if POSITION(X) > 0.0 then
            delay (DISP_TIME - SECONDS(CLOCK) - 0.02);
            ATOD.GET_VELOCITIES(VELOCITY);
            PUT_POSITION_VELOCITY(POSITION, VELOCITY(CURRENT));
            POSITION := POSITION - SIMPSON(POSITION, VELOCITY);
            DISP_TIME := DISP_TIME + INTERVAL;
        else
            NO_TARGET := TRUE;
        end if;
    end loop;
end PROJECT;

```


LIST OF REFERENCES

1. Parnas, D.L., van Schouwen, A.J., and Shu, P.K., Evaluation of Safety-Critical Software (Software Controllers in Medical and Industrial Applications), *Communications of the ACM*, June 1990
2. MIL-STD-882B Notice 1 of 1 July 1987, *System Safety Program Requirements*, U.S. Department of Defense, U.S. Government Printing Office, Washington, DC
3. Leveson, N.G., Software Safety: Why, What, and How, *ACM Computing Surveys*, June 1986
4. Gallagher, K.B. and Lyle, J.R., Software Safety and Program SLicing
5. Wetterlind, P. and Lively, W.M., Ensuring Software Safety in Robot Control, *Proceedings, 1987 Fall Joint Computer Conference, Exploring Technology: Today and Tomorrow*
6. Neumann, P.G., Regulatory Requirements for Software Safety: Policy Issues, *Proceedings of IEEE Compass '89*, Githersburg, MD, 1989
7. Leveson, N.G., Stoltzy, J.L., and Burton, B.A., Using Fault Trees to Find Design Errors in Real Time Software, *AIAA 21st Aerospace Sciences Meetings*, January 1983
8. Leveson, N.G. and Harvey, P.R., Software Fault Tree Analysis, *Journal of Systems Software* 3, 1983
9. Friedman, M.A., Automated Software Fault-Tree Analysis of Pascal Programs, *Proceedings, 1993 Annual Reliability and Maintainability Symposium*
10. Cha, S.S., Leveson, N.G., and Shimeall, T.J., Fault Tree Analysis Applied to Ada, *Proceedings of the Tenth International Conference on Software Engineering*, Singapore, 1988
11. Hansen, M.D., Survey of Available Software-Safety Analysis Techniques, *Proceeding, 1989 Annual Reliability and Maintainability Symposium*
12. Beizer, B., *Software Testing Techniques*, Van Nostrand Reinhold, New York, 1990

13. Hayward, D.L., *A Practical Application of Petri Nets in the Software Safety Analysis of a Real-Time Military System*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, December 1987
14. Gill, J.A., *Safety Analysis of Heterogenous-Multiprocessor Control System Software*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, December 1990
15. Cherniavsky, J.C., Opening Remarks, *Proceedings of the Fourth Annual Conference on Computer Assurance, Systems Integrity, Software Safety, and Process Security*
16. McGraw, R.J., *Petri Net and Fault Tree Analysis: Combining Two Techniques for a Safety Analysis on an Embedded Military Application*, M.S. Thesis, Naval Postgraduate School, Monterey, CA December 1989
17. Lewis, A.D., *Petri Net Modeling and Automated Software Safety Analysis: Methodology for an Embedded Military Application*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, June 1988
18. Ordonio, R.R., *An Automated Tool to Facilitate Code Translation for Software Fault Tree Analysis*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, September 1993
19. INMOS Reference Manual, INMOS Limited, September 1985
20. Flynn, M.J., Some Computer Organizations and Their Effectiveness, *IEEE Transactions on Computers*, Vol C-21, September 1972
21. Yuktadatta, P., *Simulation of a Parallel Processor Based Small Tactical System*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, December 1991
22. Cohen, N.H., *Ada as a Second Language*, McGraw-Hill, Inc., 1986
23. Gehani, N., *Ada Concurrent Programming*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1984
24. Redmill, F.J., *Dependability of Critical Computer Systems I*, Elsevier Science Publishing Co., Inc., New York, N.Y., 1988
25. Leveson, N.G and Turner, C.S., An Investigation of the Therac-25 Accidents, *IEEE Computer*, July 1993
26. Ada 9X Mapping/Revision Team, *ISO/IEC DIS 8652 Programming Language Ada Draft 4.4*, Intermetrics, Inc., Cambridge, MA, 1994

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2. Library, Code 52 Naval Postgraduate School Monterey, California 93943	2
3. Department Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, California 93943	1
4. Professor Timothy Shimeall, Code CS/sm Computer Science Department Naval Postgraduate School Monterey, California 93943	6
5. LCDR William S. Reid DSU DET MYSTIC (DSRV-1) P.O. Box 357049 San Diego, California 92135-7049	2
6. Dr. Michael Friedman 18950 Mt. Castile Circle Fountain Valley, CA 92708	1
7. Dr. Stephen Cha 30 Willow Tree Lane Irvine, CA 92714	1
8. Mr. Robert F. Westbrook (Code 31) Naval Air Warfare Center Weapons Division China Lake, CA 93555	1

**Best
Available
Copy**

9. Ms. Eileen T. Takach
Naval Air Warfare Center
Aircraft Division, Indpls.
DP304N MS-31
6000 E. 21st Street
Indianapolis, IN 46219-2189

1